AFRL-IF-RS-TR-2005-51
Final Technical Report
February 2005

PACE: POWER-AWARE COMPUTING ENGINES

MIT Computer Science & Artificial Intelligence Laboratory

Sponsored by
Defense Advanced Research Projects Agency
DARPA Order No. J873

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2005-51 has been reviewed and is approved for publication

APPROVED: /s/
RAYMOND A. LIUZZI
Project Engineer

FOR THE DIRECTOR: /s/
JAMES A. COLLINS, Acting Chief
Advanced Computing Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	February 2005	**FINAL** **May 00 – May 03**

4. TITLE AND SUBTITLE

PACE: POWER-AWARE COMPUTING ENGINES

5. FUNDING NUMBERS

G - F30602-00-2-0562
PE - 62301E
PR - HPSW
TA - 00
WU - 09

6. AUTHOR(S)

Krste Asanovic

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

MIT Computer Science & Artificial Intelligence Laboratory
32 Vassar Street
Cambridge MA 02139

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Defense Advanced Research Projects Agency
3701 North Fairfax Drive
Arlington VA 22203-1714

AFRL/IFT
525 Brooks Road
Rome NY 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2005-51

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Raymond A. Liuzzi/IFT/(315) 330-3577 Raymond.Liuzzi@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

This report describes the PACE project whose objective was to reduce the energy consumption of microprocessors by exploiting compile time knowledge to reduce run-time switching activity and to power down unneeded blocks. The project had two phases. The first phase focused on understanding and reducing power consumption within microprocessor components, such as caches, register files, and arithmetic units. Several new techniques were developed to reduce both switching and leakage power. The second phase developed a new energy-exposed microprocessor architecture, SCALE (Software-Controlled Architecture for Low Energy). SCALE is based on a new vector-thread architectural paradigm which unifies the vector and threaded execution models, to provide efficient execution of many forms of parallelism. The SCALE vector thread architecture and the detailed design are being pursued in other projects.

The PACE project developed a variety of power saving techniques at both the micro architectural and instruction set level, several of which are being actively transferred to industry. Over a dozen conference papers and student theses have been published to distribute results to the research community.

14. SUBJECT TERMS

Power-Aware Computing, Architecture, Hardware/Software, Compilers

15. NUMBER OF PAGES 41

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

1 Executive Summary

The aim of the PACE project was to reduce the energy consumption of microprocessors by exploiting compile-time knowledge to reduce run-time switching activity and to power down unneeded blocks. The project had two phases. The first phase focused on understanding and reducing power consumption within microprocessor components, such as caches, register files, and arithmetic units. Several new techniques were developed to reduce both switching and leakage power. The second phase developed a new *energy-exposed* microprocessor architecture, SCALE (Software-Controlled Architecture for Low Energy). SCALE is based on a new *vector-thread* architectural paradigm which unifies the vector and threaded execution models, to provide efficient execution of many forms of parallelism.

2 Approach

The project developed a highly parallel microprocessor architecture, SCALE, that is structured as an array of processing tiles. Each tile contains both processing and memory resources and the tiles communicate with each other and off-chip devices over an on-chip communications network. This tiled structure provides both high performance and low energy consumption by allowing distributed parallel computations on local data. Software can trade energy and performance by varying the number of tiles allocated to a task. In addition, each tile has an unprecedented level of fine-grain software power control to enable deactivation of unneeded microarchitectural components.

Modern instruction set architectures (ISAs), such as RISC and VLIW machines, provide a hardware-software interface designed solely for maximum performance with minimum hardware complexity. Compared with application-specific custom circuitry, these general purpose processors exhibit a factor of 100–1000 worse energy-delay product. This project worked on reducing this gap by re-examining the hardware-software interface, only now considering both performance and energy consumption. The approach was to co-develop new machine architectures that expose energy consumption to software together with new compilation technology that can communicate energy-saving compile-time knowledge to the hardware. The result was the SCALE architecture, which introduces a new vector-thread architectural paradigm that provides high performance at low power for many forms of application parallelism.

The initial phase of the project examined the power consumption in various microarchitectural components. We developed a number of power saving techniques at the microarchitectural level, and gained insight into where software could best help reduce power through the instruction set level.

To help evaluate the approach, a fast and accurate energy-performance simulation framework (SyCHOSys) was developed that enables simulation of complete microprocessor designs running large scale applications while gathering detailed energy statistics. This simulator extends the state of the art by enabling accurate ($< 10\%$ error) cycle-by-cycle energy characterization for billions of cycles of simulated CPU activity.

The compiler research in this project leveraged two existing sophisticated optimizing compiler infrastructures developed at MIT: the RAW FORTRAN and C compiler and the FLEX Java compiler. These were enhanced and extended to extract compile-time knowledge to reduce microprocessor power.

3 Accomplishments

SyCHOSys Power-Performance Simulator

We developed a compiled energy-performance simulator [1]. This simulator tracks the energy consumption for each individual signal within a processor with less than 10% error of a full SPICE-level circuit simulation, but is fast enough to simulate several billion cycles of application code in a single day on a commercial workstation.

We used the simulation to determine the energy-consumption within a complete low-power microprocessor architecture [2] running a range of application benchmarks. Results obtained illustrate areas that require further energy savings after common low-power optimizations are applied. This simulator framework was used for many of the following studies.

Activity-Sensitive Flip-Flops

Latches and flip-flops are important components of total power dissipation. We developed a new *activity-sensitive* flip-flop design methodology which reduces flip-flop and latch energy by up to 60% with no speed penalty by using detailed knowledge of the expected data and clock activity for each register [3].

We also investigated the effect of loading on flip-flop power consumption, and showed that the relative energy-delay performance of various flip-flop designs changes as both absolute output load and input-to-output load ratio are varied [4].

Cache and Register File Optimizations

In the first phase of the project, we developed a number of techniques to reduce energy in the caches and register files of processors.

Way-memorization avoids cache tag lookups by building direct links within the instruction cache. This removes 97% of instruction cache tag lookups, saving 23% of I-cache energy [5].

We developed a new *dynamic cache resizing* technique that adapts active cache size to application needs to reduce switching and leakage power in highly-associative caches. This technique typically reduces active cache size and power by one half with minimal impact on performance [6].

To reduce register file energy, we developed a banked register file scheme with a simple speculative control scheme [7]. This reduced register file size by a factor of three and access energy by 40%.

Fine-Grain Leakage Reduction

Leakage current is a growing concern as threshold voltages are scaled down. We have developed circuits and microarchitectures for fine-grain dynamic leakage reduction, which allow small portions of an active processor to be powered down for a short period of time to save static leakage power. Our techniques use *leakage-biased* circuits, where leakage currents themselves are used to bias circuits into a low-leakage state. Savings of over 57% of overall active power were estimated for a multiported register file, with no performance loss [8].

We have also developed a high-performance leakage-biased domino circuit style, which reduces standby leakage by a factor of 100 compared to dual-Vt domino [9], at the same delay.

Activity Migration

Power dissipation is distributed unevenly over the surface of a microprocessor, leading to local temperature "hot-spots", which limit sustainable power dissipation and reduce reliability.

We developed the technique of *activity migration* to reduce power density in microprocessors. Activity migration reduces die temperature by moving computation between multiple redundant circuits as each one heats up. The drop in die temperature reduces leakage current by up to 35% and increases transistor speed by up to 16% [10].

Heads-and-Tails Variable Length Instruction Encoding

We developed the heads-and-tails format, which simplifies pipelined or superscalar instruction fetch and decode of a dense variable-length instruction format. For RISC processors a 25% reduction in code size was achieved, for VLIW processors a 40% reduction in static code size was achieved [11, 12]. Reduced code size provides better hit ratios in small low-power caches.

Energy-Exposed Instruction Sets

The second phase of the project focused on how compile-time knowledge could reduce energy consumption at run time. We developed several complementary ideas in energy-exposed instruction sets [13].

Inside current microprocessors, there is considerable microarchitectural overhead in support precise exceptions on every instruction. Using *software restart markers* we can shift some of this burden to the compiler, by only marking certain instructions as requiring precise exception semantics. We implemented compiler passes in both C and Java and determined we could remove around 60% of exception points in code using only a simple local analysis [14, 13].

The compiler is responsible for register allocation, and this information can be used to reduce register file traffic. We developed a hybrid accumulator-RISC architecture that allows software to manage the bypass latches directly, and implemented compiler passes that removed up to 36% of register file reads and up to 47% of register file writes in C and Java programs [14, 13].

We also developed the direct-addressed cache, a combined hardware and software scheme that uses compile-time knowledge to remove up to 70% of data cache tag checks at run-time [15].

SCALE Vector-Thread Architecture

The SCALE architecture builds upon the experience gained in the first phase in the project. SCALE is based around an energy-exposed instruction and introduces a new architectural paradigm, *vector threading*. The vector-thread architecture unifies vector and threaded parallel execution models to give high performance on a wide range of applications [16].

An instruction-level simulator and a detailed microarchitectural-level cycle simulator have been completed for SCALE.

We are continuing to complete a prototype implementation of the SCALE architecture in other work.

Mondriaan Memory Protection

A new fine-grained memory protection system, *Mondriaan Memory protection*, was developed as an offshoot of the software-controlled low-power cache design [17, 18, 19]. This scheme provides efficient hardware memory protection to improve system robustness.

A patent has been filed for this technique.

4 Technology transition

Numerous technology transition paths are being pursued to transfer results to industrial partners.

Activity-sensitive Flip-Flops and Latches

The activity-sensitive flip-flop and latch methodology has been transferred to the Desktop Products Group at Intel Corporation, where it was evaluated and cleared for use in product development.

Heads and Tails Instruction Compression

A collaboration with Paolo Faraboschi and Josh Fisher at HP laboratories was undertaken to evaluate Heads-and-Tails instruction encoding for HP's Lx embedded VLIW microprocessor, using HP compilers and simulators.

Fine-Grain Dynamic Leakage Reduction

Fine-Grain Dynamic Leakage Reduction Techniques for fine-grain dynamic leakage reduction are being evaluated within the Desktop Products Group at Intel Corporation. An MIT graduate student worked as an intern with George Cai at Intel, Austin to help with technology transition. Intel is continuing to fund this work at MIT.

Banked Register Files

A graduate student is currently working with Xiaowie Chen at IBM T. J. Watson evaluating the use of banked register files within future IBM PowerPC processors.

Power Modeling

A detailed cache and memory energy model, ZOOM, was developed in collaboration with Jude Rivers at IBM's T.J. Watson Laboratory. A student worked at IBM for the summer to incorporate data from commercial cache designs.

A second graduate student is currently working on power models for single-chip multiprocessors with Pradip Bose at IBM T. J. Watson.

5 Conclusion

A variety of power saving techniques at both the microarchitectural and instruction set level have been developed, several of which are being actively transferred to industry through student internships. Over a dozen conference papers and student theses have been published to distribute results to the research community. The SCALE vector-thread architecture was developed and the detailed design is now being pursued in other work.

6 References

[1] R. Krashinsky, S. Heo, M. Zhang, and K. Asanović. SyCHOSys: Compiled energy-performance cycle simulation. In *Workshop on Complexity-Effective Design, 27th ISCA*, Vancouver, Canada, June 2000.

[2] R. Krashinsky. Microprocessor energy characterization and optimization through fast, accurate, and flexible simulation. Master's thesis, Massachusetts Institute of Technology, May 2001.

[3] S. Heo, R. Krashinsky, and K. Asanović. Activity-sensitive flip-flop and latch selection for reduced energy. In *19th Conference on Advanced Research in VLSI*, Salt Lake City,UT USA, March 2001.

[4] S. Heo and K. Asanović. Load-sensitive flip-flop characterization. In *IEEE Workshop on VLSI*, Orlando, FL, April 2001.

[5] A. Ma, M. Zhang, and K. Asanović. Way memoization to reduce fetch energy in instruction caches. *Workshop on Complexity-Effective Design, International Symposium on Computer Architecture*, June 2001.

[6] M. Zhang and K. Asanović. Miss tags for fine-grain CAM-tag cache resizing. In *International Symposium on Low Power Electronics and Design*, Monterey, CA, August 2002.

[7] J. Tseng and K. Asanović. Banked multiported register files for high-frequency superscalar microprocessors. In *30th International Symposium on Computer Architecture*, San Diego, CA, June 2003.

[8] S. Heo, K. Barr, M. Hampton, and K. Asanović. Dynamic fine-grain leakage reduction using leakage-biased bitlines. In *International Symposium on Computer Architecture*, Anchorage, AK, May 2002.

[9] S. Heo and K. Asanović. Leakage-biased domino circuits for dynamic fine-grain leakage reduction. In *Symposium on VLSI Circuits*, Honolulu, HI, June 2002.

[10] S. Heo, K. Barr, and K. Asanović. Reducing power density through activity migration. In *International Symposium on Low Power Electronics and Design*, Seoul, Korea, August 2003.

[11] H. Pan and K. Asanović. Heads and Tails: A variable-length instruction format supporting parallel fetch and decode. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, Atlanta, GA, November 2001.

[12] H. Pan. High-performance variable-length instruction encodings. Master's thesis, Massachusetts Institute of Technology, May 2002.

[13] K. Asanović, M. Hampton, R. Krashinsky, and E. Witchel. Energy-exposed instruction sets. In R. Graybill and R. Melhem, editors, *Power-Aware Computing*. Kluwer/Plenum Publishing, 2002.

[14] M. Hampton. Exposing datapath elements to reduce microprocessor energy consumption. Master's thesis, Massachusetts Institute of Technology, June 2001.

[15] E. Witchel, S. Larsen, C. S. Ananian, and K. Asanović. Direct addressed caches for reduced power consumption. In *34th International Symposium on Microarchitecture*, Austin, TX, December 2001.

[16] R. Krashinsky, C. Batten, S. Gerding, M. Hampton, B. Pharris, J. Casper, and K. Asanović. The vector-thread architecture. In *31st International Symposium on Computer Architecture*, Munich, Germany, June 2004.

[17] E. Witchel, J. Cates, and K. Asanović. Mondrian memory protection. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 304–316, San Jose, CA, October 2002.

[18] E. Witchel and K. Asanović. Hardware works, software doesn't: Enforcing modularity with Mondriaan memory protection. In *Ninth Workshop on Hot Topics in Operating Systems*, Lihue, HI, May 2003.

[19] E. Witchel. *Mondriaan Memory Protection*. PhD thesis, Massachusetts Institute of Technology, 2004.

APPENDIX A - The Vector-Thread Architecture

Ronny Krashinsky, Christopher Batten, Mark Hampton, Steve Gerding,
Brian Pharris, Jared Casper, and Krste Asanović
MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge, MA 02139
{ronny,cbatten,krste}@csail.mit.edu

Abstract

The vector-thread (VT) architectural paradigm unifies the vector and multithreaded compute models. The VT abstraction provides the programmer with a control processor and a vector of virtual processors (VPs). The control processor can use vector-fetch commands to broadcast instructions to all the VPs or each VP can use thread-fetches to direct its own control flow. A seamless intermixing of the vector and threaded control mechanisms allows a VT architecture to flexibly and compactly encode application parallelism and locality, and a VT machine exploits these to improve performance and efficiency. We present SCALE, an instantiation of the VT architecture designed for low-power and high-performance embedded systems. We evaluate the SCALE prototype design using detailed simulation of a broad range of embedded applications and show that its performance is competitive with larger and more complex processors.

1. Introduction

Parallelism and locality are the key application characteristics exploited by computer architects to make productive use of increasing transistor counts while coping with wire delay and power dissipation. Conventional sequential ISAs provide minimal support for encoding parallelism or locality, so high-performance implementations are forced to devote considerable area and power to on-chip structures that extract parallelism or that support arbitrary global communication. The large area and power overheads are justified by the demand for even small improvements in performance on legacy codes for popular ISAs. Many important applications have abundant parallelism, however, with dependencies and communication patterns that can be statically determined. ISAs that expose more parallelism reduce the need for area and power intensive structures to extract dependencies dynamically. Similarly, ISAs that allow locality to be expressed reduce the need for long-range communication and complex interconnect. The challenge is to develop an efficient encoding of an application's parallel dependency graph and to reduce the area and power consumption of the microarchitecture that will execute this dependency graph.

In this paper, we unify the vector and multithreaded execution models with the *vector-thread* (VT) architectural paradigm. VT allows large amounts of structured parallelism to be compactly encoded in a form that allows a simple microarchitecture to attain high performance at low power by avoiding complex control and datapath structures and by reducing activity on long wires. The VT programmer's model extends a conventional scalar control processor with an array of slave *virtual processors* (VPs). VPs execute strings of RISC-like instructions packaged into *atomic instruction blocks* (AIBs). To execute data-parallel code, the control processor broadcasts AIBs to all the slave VPs. To execute thread-parallel code, each VP directs its own control flow by fetching its own AIBs. Implementations of the VT architecture can also exploit instruction-level parallelism within AIBs.

In this way, the VT architecture supports a modeless intermingling of all forms of application parallelism. This flexibility provides new ways to parallelize codes that are difficult to vectorize or that incur excessive synchronization costs when threaded. Instruction locality is improved by allowing common code to be factored out and executed only once on the control processor, and by executing the same AIB multiple times on each VP in turn. Data locality is improved as most operand communication is isolated to within an individual VP.

We are developing a prototype processor, SCALE, which is an instantiation of the vector-thread architecture designed for low-power and high-performance embedded systems. As transistors have become cheaper and faster, embedded applications have evolved from simple control functions to cellphones that run multitasking networked operating systems with realtime video, three-dimensional graphics, and dynamic compilation of garbage-collected languages. Many other embedded applications require sophisticated high-performance information processing, including streaming media devices, network routers, and wireless base stations. In this paper, we show how benchmarks taken from these embedded domains can be mapped efficiently to the SCALE vector-thread architecture. In many cases, the codes exploit multiple types of parallelism simultaneously for greater efficiency.

The paper is structured as follows. Section 2 introduces the vector-thread architectural paradigm. Section 3 then describes the SCALE processor which contains many features that extend the basic VT architecture. Section 4 presents an evaluation of the SCALE processor using a range of embedded benchmarks and describes how SCALE efficiently executes various types of code. Finally, Section 5 reviews related work and Section 6 concludes.

2. The VT Architectural Paradigm

An architectural paradigm consists of the programmer's model for a class of machines plus the expected structure of implementations of these machines. This section first describes the abstraction a VT architecture provides to a programmer, then gives an overview of the physical model for a VT machine.

2.1 VT Abstract Model

The vector-thread architecture is a hybrid of the vector and multithreaded models. A conventional *control processor* interacts with a *virtual processor vector* (VPV), as shown in Figure 1. The programming model consists of two interacting instruction sets, one for the control processor and one for the VPs. Applications can be mapped to the VT architecture in a variety of ways but it is es-

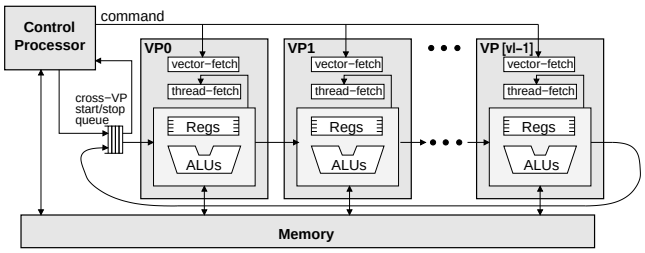

Figure 1: Abstract model of a vector-thread architecture. A control processor interacts with a virtual processor vector (an ordered sequence of VPs).

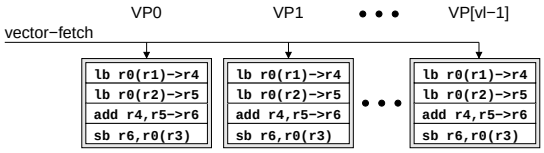

Figure 2: Vector-fetch commands. For simple data-parallel loops, the control processor can use a vector-fetch command to send an atomic instruction block (AIB) to all the VPs in parallel. In this vector-vector add example, we assume that `r0` has been loaded with each VP's index number; and `r1`, `r2`, and `r3` contain the base addresses of the input and output vectors. The instruction notation places the destination registers after the "`->`".

pecially well suited to executing loops; each VP executes a single iteration of the loop and the control processor is responsible for managing the execution.

A virtual processor contains a set of registers and has the ability to execute RISC-like instructions with virtual register specifiers. VP instructions are grouped into *atomic instruction blocks* (AIBs), the unit of work issued to a VP at one time. There is no automatic program counter or implicit instruction fetch mechanism for VPs; all instruction blocks must be explicitly requested by either the control processor or the VP itself.

The control processor can direct the VPs' execution using a *vector-fetch* command to issue an AIB to all the VPs in parallel, or a *VP-fetch* to target an individual VP. Vector-fetch commands provide a programming model similar to conventional vector machines except that a large block of instructions can be issued at once. As a simple example, Figure 2 shows the mapping for a data-parallel vector-vector add loop. The AIB for one iteration of the loop contains two loads, an add, and a store. A vector-fetch command sends this AIB to all the VPs in parallel and thus initiates `vl` loop iterations, where `vl` is the length of the VPV (i.e., the vector length). Every VP executes the same instructions but operates on distinct data elements as determined by its index number. As a more efficient alternative to the individual VP loads and stores shown in the example, a VT architecture can also provide vector-memory commands issued by the control processor which move a vector of elements between memory and one register in each VP.

The VT abstract model connects VPs in a unidirectional ring topology and allows a sending instruction on VP (n) to transfer data directly to a receiving instruction on VP ($n+1$). These *cross-VP data transfers* are dynamically scheduled and resolve when the data becomes available. Cross-VP data transfers allow loops with cross-iteration dependencies to be efficiently mapped to the vector-thread architecture, as shown by the example in Figure 3. A single vector-fetch command can introduce a chain of `prevVP` receives and `nextVP` sends that spans the VPV. The control processor can push an initial value into the *cross-VP start/stop queue* (shown in Figure 1) before executing the vector-fetch command. After the chain executes, the final cross-VP data value from the last VP wraps

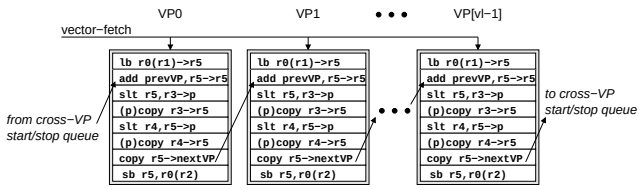

Figure 3: Cross-VP data transfers. For loops with cross-iteration dependencies, the control processor can vector-fetch an AIB that contains cross-VP data transfers. In this saturating parallel prefix sum example, we assume that `r0` has been loaded with each VP's index number, `r1` and `r2` contain the base addresses of the input and output vectors, and `r3` and `r4` contain the min and max values of the saturation range. The instruction notation uses "`(p)`" to indicate predication.

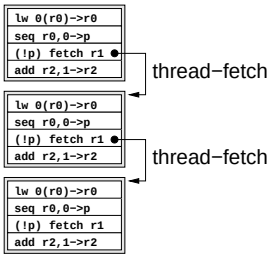

Figure 4: VP threads. Thread-fetches allow a VP to request its own AIBs and thereby direct its own control flow. In this pointer-chase example, we assume that `r0` contains a pointer to a linked list, `r1` contains the address of the AIB, and `r2` contains a count of the number of links traversed.

around and is written into this same queue. It can then be popped by the control processor or consumed by a subsequent `prevVP` receive on VP0 during stripmined loop execution.

The VT architecture also allows VPs to direct their own control flow. A VP executes a *thread-fetch* to request an AIB to execute after it completes its active AIB, as shown in Figure 4. Fetch instructions may be predicated to provide conditional branching. A VP thread persists as long as each AIB contains an executed fetch instruction, but halts once the VP stops issuing thread-fetches. Once a VP thread is launched, it executes to completion before the next command from the control processor takes effect. The control processor and VPs all operate concurrently in the same address space. Memory dependencies between these processors are preserved via explicit memory fence and synchronization operations or atomic read-modify-write operations.

The ability to freely intermix vector-fetches and thread-fetches allows a VT architecture to combine the best attributes of the vector and multithreaded execution paradigms. As shown in Figure 5, the control processor can issue a vector-fetch command to launch a vector of VP threads, each of which continues to execute as long as it issues thread-fetches. These thread-fetches break the rigid control flow of traditional vector machines, enabling the VP threads to follow independent control paths. Thread-fetches broaden the range of loops which can be mapped efficiently to VT, allowing the VPs to execute data-parallel loop iterations with conditionals or even inner-loops. Apart from loops, the VPs can also be used as *free-running threads*, where they operate independently from the control processor and retrieve tasks from a shared work queue.

The VT architecture allows software to efficiently expose structured parallelism and locality at a fine granularity. Compared to a conventional threaded architecture, the VT model allows common bookkeeping code to be factored out and executed once on the control processor rather than redundantly in each thread. AIBs enable a VT machine to efficiently amortize instruction fetch overhead, and provide a framework for cleanly handling temporary

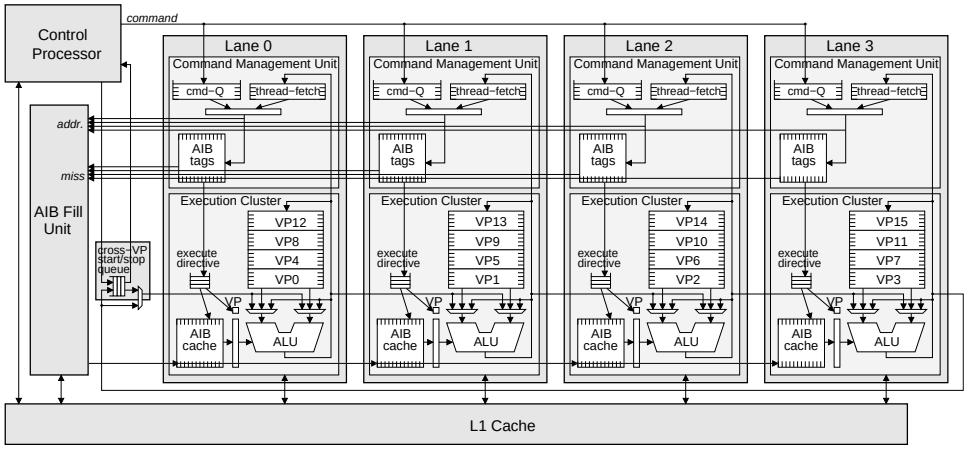

Figure 6: Physical model of a VT machine. The implementation shown has four parallel lanes in the vector-thread unit (VTU), and VPs are striped across the lane array with the low-order bits of a VP index indicating the lane to which it is mapped. The configuration shown uses VPs with five virtual registers, and with twenty physical registers each lane is able to support four VPs. Each lane is divided into a command management unit (CMU) and an execution cluster, and the execution cluster has an associated cross-VP start-stop queue.

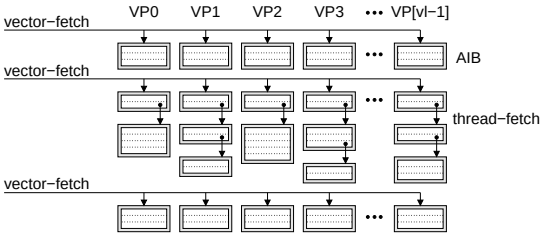

Figure 5: The control processor can use a vector-fetch command to send an AIB to all the VPs, after which each VP can use thread-fetches to fetch its own AIBs.

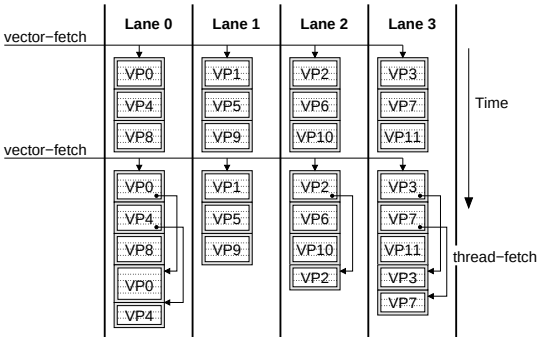

Figure 7: Lane Time-Multiplexing. Both vector-fetched and thread-fetched AIBs are time-multiplexed on the physical lanes.

state. Vector-fetch commands explicitly encode parallelism and instruction locality, allowing a VT machine to attain high performance while amortizing control overhead. Vector-memory commands avoid separate load and store requests for each element, and can be used to exploit memory data-parallelism even in loops with non-data-parallel compute. For loops with cross-iteration dependencies, cross-VP data transfers explicitly encode fine-grain communication and synchronization, avoiding heavyweight inter-thread memory coherence and synchronization primitives.

2.2 VT Physical Model

An architectural paradigm's physical model is the expected structure for efficient implementations of the abstract model. The VT physical model contains a conventional scalar unit for the control processor together with a *vector-thread unit* (VTU) that executes the VP code. To exploit the parallelism exposed by the VT abstract model, the VTU contains a parallel array of processing *lanes* as shown in Figure 6. Lanes are the physical processors which VPs map onto, and the VPV is striped across the lane array. Each lane contains physical registers, which hold the state of VPs mapped to the lane, and functional units, which are time-multiplexed across the VPs. In contrast to traditional vector machines, the lanes in a VT machine execute decoupled from each other. Figure 7 shows an abstract view of how VP execution is time-multiplexed on the lanes for both vector-fetched and thread-fetched AIBs. This fine-grain interleaving helps VT machines hide functional unit, memory, and thread-fetch latencies.

As shown in Figure 6, each lane contains both a *command management unit* (CMU) and an *execution cluster*. An execution cluster consists of a register file, functional units, and a small AIB cache. The lane's CMU buffers commands from the control processor in a queue (cmd-Q) and holds pending thread-fetch addresses for the lane's VPs. The CMU also holds the tags for the lane's AIB cache. The AIB cache can hold one or more AIBs and must be at least large enough to hold an AIB of the maximum size defined in the VT architecture.

The CMU chooses a vector-fetch, VP-fetch, or thread-fetch command to process. The fetch command contains an address which is looked up in the AIB tags. If there is a miss, a request is sent to the fill unit which retrieves the requested AIB from the primary cache. The fill unit handles one lane's AIB miss at a time, except if lanes are executing vector-fetch commands when refill overhead is amortized by broadcasting the AIB to all lanes simultaneously.

After a fetch command hits in the AIB cache or after a miss refill has been processed, the CMU generates an *execute directive* which contains an index into the AIB cache. For a vector-fetch command the execute directive indicates that the AIB should be executed by all VPs mapped to the lane, while for a VP-fetch or thread-fetch command it identifies a single VP to execute the AIB. The execute directive is sent to a queue in the execution cluster, leaving the CMU free to begin processing the next command. The CMU is able to overlap the AIB cache refill for new fetch commands with the execution of previous ones, but must track which AIBs have outstanding execute directives to avoid overwriting their entries in the AIB cache. The CMU must also ensure that the VP threads execute to completion before initiating a subsequent vector-fetch.

To process an execute directive, the cluster reads VP instructions8

one by one from the AIB cache and executes them for the appropriate VP. When processing an execute-directive from a vector-fetch command, all of the instructions in the AIB are executed for one VP before moving on to the next. The virtual register indices in the VP instructions are combined with the active VP number to create an index into the physical register file. To execute a fetch instruction, the cluster sends the requested AIB address to the CMU where the VP's associated pending thread-fetch register is updated.

The lanes in the VTU are inter-connected with a unidirectional ring network to implement the cross-VP data transfers. When a cluster encounters an instruction with a `prevVP` receive, it stalls until the data is available from its predecessor lane. When the VT architecture allows multiple cross-VP instructions in a single AIB, with some sends preceding some receives, the hardware implementation must provide sufficient buffering of send data to allow all the receives in an AIB to execute. By induction, deadlock is avoided if each lane ensures that its predecessor can never be blocked trying to send it cross-VP data.

3. The SCALE VT Architecture

SCALE is an instance of the VT architectural paradigm designed for embedded systems. The SCALE architecture has a MIPS-based control processor extended with a VTU. The SCALE VTU aims to provide high performance at low power for a wide range of applications while using only a small area. This section describes the SCALE VT architecture, presents a simple code example implemented on SCALE, and gives an overview of the SCALE microarchitecture and SCALE processor prototype.

3.1 SCALE Extensions to VT

Clusters

To improve performance while reducing area, energy and circuit delay, SCALE extends the single-cluster VT model (shown in Figure 1) by partitioning VPs into multiple execution clusters with independent register sets. VP instructions target an individual cluster and perform RISC-like operations. Source operands must be local to the cluster, but results can be written to any cluster in the VP, and an instruction can write its result to multiple destinations. Each cluster within a VP has a separate predicate register, and instructions can be positively or negatively predicated.

SCALE clusters are heterogeneous, but all clusters support basic integer operations. Cluster 0 additionally supports memory access instructions, cluster 1 supports fetch instructions, and cluster 3 supports integer multiply and divide. Though not used in this paper, the SCALE architecture allows clusters to be enhanced with layers of additional functionality (e.g., floating-point operations, fixed-point operations, and sub-word SIMD operations), or new clusters to be added to perform specialized operations.

Registers and VP Configuration

The general registers in each cluster of a VP are categorized as either *private registers* (`pr`'s) and *shared registers* (`sr`'s). Both private and shared registers can be read and written by VP instructions and by commands from the control processor. The main difference is that private registers preserve their values between AIBs, while shared registers may be overwritten by a different VP. Shared registers can be used as temporary state within an AIB to increase the number of VPs that can be supported by a fixed number of physical registers. The control processor can also vector-write the shared registers to broadcast scalar values and constants used by all VPs.

In addition to the general registers, each cluster also has programmer-visible *chain registers* (`cr0` and `cr1`) associated with the two ALU input operands. These can be used as sources and destinations to avoid reading and writing the register files. Like shared registers, chain registers may be overwritten between AIBs, and they are also implicitly overwritten when a VP instruction uses their associated operand position. Cluster 0 has a special chain register called the store-data (`sd`) register through which all data for VP stores must pass.

In the SCALE architecture, the control processor configures the VPs by indicating how many shared and private registers are required in each cluster. The length of the virtual processor vector changes with each re-configuration to reflect the maximum number of VPs that can be supported. This operation is typically done once outside each loop, and state in the VPs is undefined across reconfigurations. Within a lane, the VTU maps shared VP registers to shared physical registers. Control processor vector-writes to a shared register are broadcast to each lane, but individual VP writes to a shared register are not coherent across lanes, i.e., the shared registers are not *global* registers.

Vector-Memory Commands

In addition to VP load and store instructions, SCALE defines vector-memory commands issued by the control processor for efficient structured memory accesses. Like vector-fetch commands, these operate across the virtual processor vector; a vector-load writes the load data to a private register in each VP, while a vector-store reads the store data from a private register in each VP. SCALE also supports vector-load commands which target shared registers to retrieve values used by all VPs. In addition to the typical unit-stride and strided vector-memory access patterns, SCALE provides vector segment accesses where each VP loads or stores several contiguous memory elements to support "array-of-structures" data layouts efficiently.

3.2 SCALE Code Example

This section presents a simple code example to show how SCALE is programmed. The C code in Figure 8 implements a simplified version of the ADPCM speech decoder. Input is read from a unit-stride byte stream and output is written to a unit-stride halfword stream. The loop is non-vectorizable because it contains two loop-carried dependencies: the `index` and `valpred` variables are accumulated from one iteration to the next with saturation. The loop also contains two table lookups.

The SCALE code to implement the example decoder function is shown in Figure 9. The code is divided into two sections with MIPS control processor code in the `.text` section and SCALE VP code in the `.sisa` (SCALE ISA) section. The SCALE VP code implements one iteration of the loop with a single AIB; cluster 0 accesses memory, cluster 1 accumulates `index`, cluster 2 accumulates `valpred`, and cluster 3 does the multiply.

The control processor first configures the VPs using the `vcfgvl` command to indicate the register requirements for each cluster. In this example, c0 uses one private register to hold the input data and two shared registers to hold the table pointers; c1 and c2 each use three shared registers to hold the min and max saturation values and a temporary; c2 also uses a private register to hold the output value; and c3 uses only chain registers so it does not need any shared or private registers. The configuration indirectly sets `vlmax`, the maximum vector length. In a SCALE implementation with 32 physical registers per cluster and four lanes, `vlmax` would be: $(\lfloor(32-3)/1\rfloor) \times 4 = 116$, limited by the register demands of cluster 2. The `vcfgvl` command also sets `vl`, the active vector-length, to the minimum of `vlmax` and the length argument provided; the resulting length is returned as a result. The control pro-

```
void decode_ex(int len, u_int8_t* in, int16_t* out) {
  int i;
  int index = 0;
  int valpred = 0;
  for(i = 0; i < len; i++) {
    u_int8_t delta = in[i];
    index += indexTable[delta];
    index = index < IX_MIN ? IX_MIN : index;
    index = IX_MAX < index ? IX_MAX : index;
    valpred += stepsizeTable[index] * delta;
    valpred = valpred < VALP_MIN ? VALP_MIN : valpred;
    valpred = VALP_MAX < valpred ? VALP_MAX : valpred;
    out[i] = valpred;
  }
}
```

Figure 8: C code for decoder example.

cessor next vector-writes several shared VP registers with constants using the `vwrsh` command, then uses the `xvppush` command to push the initial `index` and `valpred` values into the cross-VP start/stop queues for clusters 1 and 2.

The ISA for a VT architecture is defined so that code can be written to work with any number of VPs, allowing the same object code to run on implementations with varying or configurable resources. To manage the execution of the loop, the control processor uses *stripmining* to repeatedly launch a vector of loop iterations. For each iteration of the stripmine loop, the control processor uses the `setvl` command which sets the vector-length to the minimum of `vlmax` and the length argument provided (i.e., the number of iterations remaining for the loop); the resulting vector-length is also returned as a result. In the decoder example, the control processor then loads the input using an auto-incrementing vector-load-byte-unsigned command (`vlbuai`), vector-fetches the AIB to compute the decode, and stores the output using an auto-incrementing vector-store-halfword command (`vshai`). The cross-iteration dependencies are passed from one stripmine vector to the next through the cross-VP start/stop queues. At the end of the function the control processor uses the `xvppop` command to pop and discard the final values.

The SCALE VP code implements one iteration of the loop in a straightforward manner with no cross-iteration static scheduling. Cluster 0 holds the `delta` input value in `pr0` from the previous vector-load. It uses a VP load to perform the `indexTable` lookup and sends the result to cluster 1. Cluster 1 uses five instructions to accumulate and saturate `index`, using `prevVP` and `nextVP` to receive and send the cross-iteration value, and the `psel` (predicate-select) instruction to optimize the saturation. Cluster 0 then performs the `stepsizeTable` lookup using the index value, and sends the result to cluster 3 where it is multiplied by `delta`. Cluster 2 uses five instructions to accumulate and saturate `valpred`, writing the result to `pr0` for the subsequent vector-store.

```
  .text  # MIPS control processor code
decode_ex: # a0=len, a1=in, a2=out
  # configure VPs: c0:p,s c1:p,s c2:p,s c3:p,s
  vcfgvl  t1, a0,     1,2,   0,3,   1,3,   0,0
  # (vl,t1) = min(a0,vlmax)
  sll     t1, t1, 1       # output stride
  la      t0, indexTable
  vwrsh   t0, c0/sr0      # indexTable addr.
  la      t0, stepsizeTable
  vwrsh   t0, c0/sr1      # stepsizeTable addr.
  vwrsh   IX_MIN, c1/sr0  # index min
  vwrsh   IX_MAX, c1/sr1  # index max
  vwrsh   VALP_MIN, c2/sr0# valpred min
  vwrsh   VALP_MAX, c2/sr1# valpred max
  xvppush $0, c1          # push initial index = 0
  xvppush $0, c2          # push initial valpred = 0
stripmineloop:
  setvl   t2, a0          # (vl,t2) = min(a0,vlmax)
  vlbuai  a1, t2, c0/pr0  # vector-load input, inc ptr
  vf      vtu_decode_ex   # vector-fetch AIB
  vshai   a2, t1, c2/pr0  # vector-store output, inc ptr
  subu    a0, t2          # decrement count
  bnez    a0, stripmineloop # loop until done
  xvppop  $0, c1          # pop final index, discard
  xvppop  $0, c2          # pop final valpred, discard
  vsync                   # wait until VPs are done
  jr ra                   # return

  .sisa  # SCALE VP code
vtu_decode_ex:
  .aib begin
  c0 sll  pr0, 2   -> cr1            # word offset
  c0 lw   cr1(sr0) -> c1/cr0         # load index
  c0 copy pr0      -> c3/cr0         # copy delta
  c1 addu cr0, prevVP -> cr0         # accum. index
  c1 slt  cr0, sr0 -> p              # index min
  c1 psel cr0, sr0 -> sr2            # index min
  c1 slt  sr1, sr2 -> p              # index max
  c1 psel sr2, sr1 -> c0/cr0, nextVP # index max
  c0 sll  cr0, 2   -> cr1            # word offset
  c0 lw   cr1(sr1) -> c3/cr1         # load step
  c3 mult.lo cr0, cr1 -> c2/cr0      # step*delta
  c2 addu cr0, prevVP -> cr0         # accum. valpred
  c2 slt  cr0, sr0 -> p              # valpred min
  c2 psel cr0, sr0 -> sr2            # valpred min
  c2 slt  sr1, sr2 -> p              # valpred max
  c2 psel sr2, sr1 -> pr0, nextVP    # valpred max
  .aib end
```

Figure 9: SCALE code implementing decoder example from Figure 8.

3.3 SCALE Microarchitecture

The SCALE microarchitecture is an extension of the general VT physical model shown in Figure 6. A lane has a single CMU and one physical execution cluster per VP cluster. Each cluster has a dedicated output bus which broadcasts data to the other clusters in the lane, and it also connects to its sibling clusters in neighboring lanes to support cross-VP data transfers. An overview of the SCALE lane microarchitecture is shown in Figure 10.

Micro-Ops and Cluster Decoupling

The SCALE software ISA is portable across multiple SCALE implementations, but is designed to be easy to translate into implementation-specific micro-operations, or *micro-ops*. The assembler translates the SCALE software ISA into the native hardware ISA at compile time. There are three categories of hardware micro-ops: a *compute-op* performs the main RISC-like operation of a VP instruction; a *transport-op* sends data to another cluster; and, a *writeback-op* receives data sent from an external cluster. The assembler reorganizes micro-ops derived from an AIB into *micro-op bundles* which target a single cluster and do not access other clusters' registers. Figure 11 shows how the SCALE VP instructions from the decoder example are translated into micro-op bundles. All inter-cluster data dependencies are encoded by the transport-ops and writeback-ops which are added to the sending and receiving cluster respectively. This allows the micro-op bundles for each cluster to be packed together independently from the micro-op bundles for other clusters.

Partitioning inter-cluster data transfers into separate transport and writeback operations enables decoupled execution between clusters. In SCALE, a cluster's AIB cache contains micro-op bundles, and each cluster has a local execute directive queue and local control. Each cluster processes its transport-ops in order, broadcasting result values onto its dedicated output data bus; and each cluster processes its writeback-ops in order, writing the values from external clusters to its local registers. The inter-cluster data dependencies are synchronized with handshake signals which extend between the clusters, and a transaction only completes when both the

Cluster 0			Cluster 1			Cluster 2			Cluster 3		
wb-op	compute-op	tp-op	wb-op	compute-op	tp-op	wb-op	compute-op	tp-op	wb-op	compute-op	tp-op
	sll pr0,2→cr1		*b*,c0→cr0	addu cr0,pVP→cr0		*b*,c3→cr0	addu cr0,pVP→cr0		*b*,c0→cr0		
	lw cr1(sr0)	→c1		slt cr0,sr0→p			slt cr0,sr0→p		*b*,c0→cr1	mult cr0,cr1	→c2
c1→cr0	copy pr0	→c3		psel cr0,sr0→sr2			psel cr0,sr0→sr2				
	sll cr0,2→cr1			slt sr1,sr2→p			slt sr1,sr2→p				
	lw cr1(sr1)	→c3		psel sr2,sr1	→nVP,c0		psel sr2,sr1→pr0	→nVP			

Figure 11: Cluster micro-op bundles for the AIB in Figure 9. The writeback-op field is labeled as 'wb-op' and the transport-op field is labeled as 'tp-op'. A writeback-op is marked with '*b*' when the dependency order is writeback-op followed by compute-op. The prevVP and nextVP identifiers are abbreviated as 'pVP' and 'nVP'.

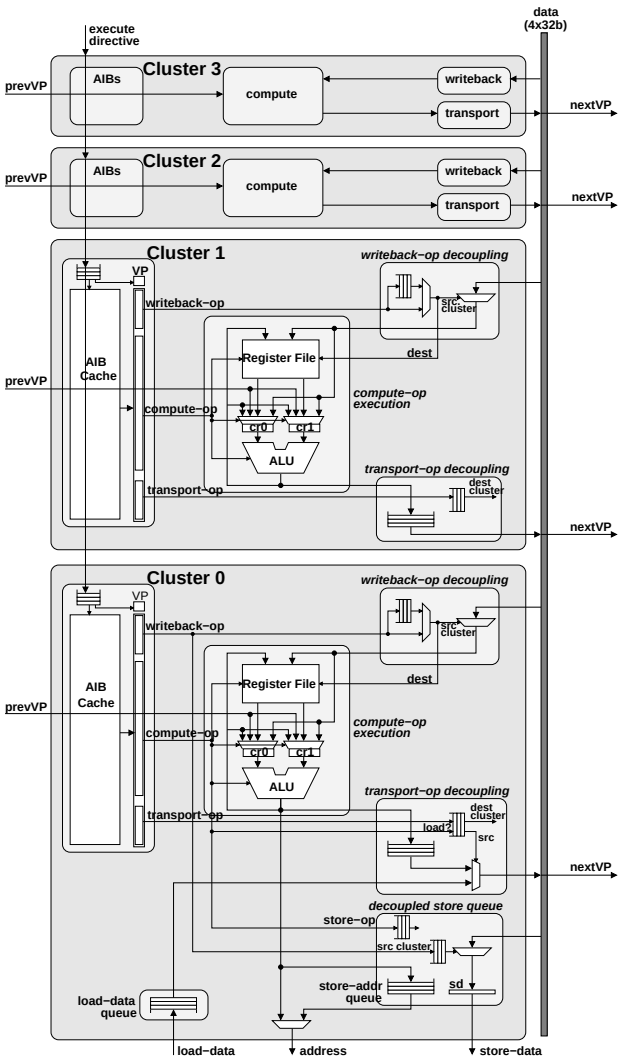

Figure 10: SCALE Lane Microarchitecture. The AIB caches in SCALE hold micro-op bundles. The compute-op is a local RISC operation on the cluster, the transport-op sends data to external clusters, and the writeback-op receives data from external clusters. Clusters 1, 2, and 3 are basic cluster designs with writeback-op and transport-op decoupling resources (cluster 1 is shown in detail, clusters 2 and 3 are shown in abstract). Cluster 0 connects to memory and includes memory access decoupling resources.

sender and the receiver are ready. Although compute-ops execute in order, each cluster contains a transport queue to allow execution to proceed without waiting for external destination clusters to receive the data, and a writeback queue to allow execution to proceed without waiting for data from external clusters (until it is needed by a compute-op). These queues make inter-cluster synchronization more flexible, and thereby enhance cluster decoupling.

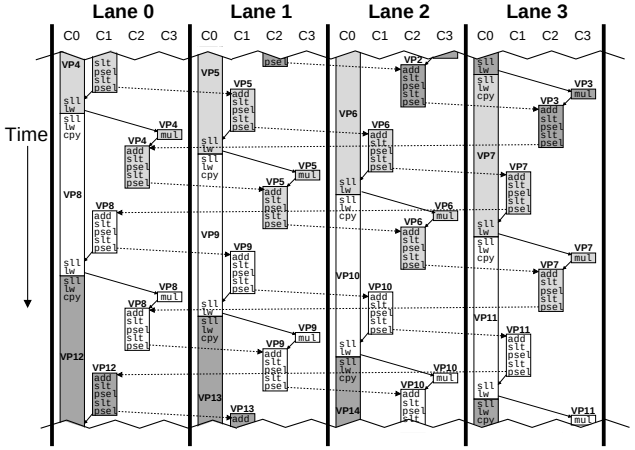

Figure 12: Execution of decoder example on SCALE. Each cluster executes in-order, but cluster and lane decoupling allows the execution to automatically adapt to the software critical path. Critical dependencies are shown with arrows (solid for inter-cluster within a lane, dotted for cross-VP).

A schematic diagram of the example decoder loop executing on SCALE (extracted from simulation trace output) is shown in Figure 12. Each cluster executes the vector-fetched AIB for each VP mapped to its lane, and decoupling allows each cluster to advance to the next VP independently. Execution automatically adapts to the software critical path as each cluster's local data dependencies resolve. In the example loop, the accumulations of `index` and `valpred` must execute serially, but all of the other instructions are not on the software critical path. Furthermore, the two accumulations can execute in parallel, so the cross-iteration serialization penalty is paid only once. Each loop iteration (i.e., VP) executes over a period of 30 cycles, but the combination of multiple lanes and cluster decoupling within each lane leads to as many as six loop iterations executing simultaneously.

Memory Access Decoupling

All VP loads and stores execute on cluster 0 (c0), and it is specially designed to enable *access-execute decoupling* [11]. Typically, c0 loads data values from memory and sends them to other clusters, computation is performed on the data, and results are returned to c0 and stored to memory. With basic cluster decoupling, c0 can continue execution after a load without waiting for the other clusters to receive the data. Cluster 0 is further enhanced to hide memory latencies by continuing execution after a load misses in the cache, and therefore it may retrieve load data from the cache out of order. However, like other instructions, load operations on cluster 0 use transport-ops to deliver data to other clusters in order, and c0 uses a *load data queue* to buffer the data and preserve the correct ordering.

Importantly, when cluster 0 encounters a store, it does not stall to

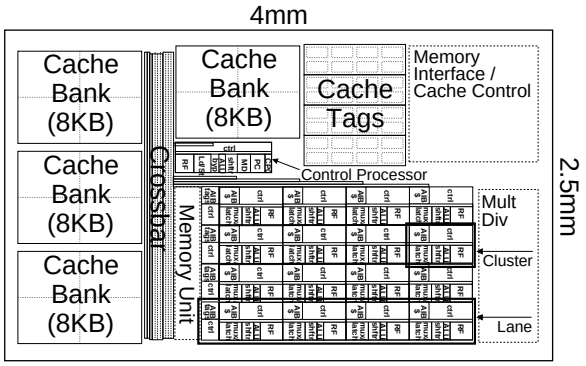

Figure 13: Preliminary floorplan estimate for SCALE prototype. The prototype contains a scalar control processor, four 32-bit lanes with four execution clusters each, and 32 KB of cache in an estimated 10 mm^2 in 0.18 μm technology.

wait for the data to be ready. Instead it buffers the store operation, including the store address, in the *decoupled store queue* until the store data is available. When a SCALE VP instruction targets the `sd` register, the resulting transport-op sends data to the store unit rather than to c0; thus, the store unit acts as a primary destination for inter-cluster transport operations and it handles the writeback-ops for `sd`. Store decoupling allows a lane's load stream to slip ahead of its store stream, but loads for a given VP are not allowed to bypass previous stores to the same address by the same VP.

Vector-Memory Accesses

Vector-memory commands are sent to the clusters as special execute directives which generate micro-ops instead of reading them from the AIB cache. For a vector-load, writeback-ops on the destination cluster receive the load data; and for a vector-store, compute-ops and transport-ops on the source cluster read and send the store data. *Chaining* is provided to allow overlapped execution of vector-fetched AIBs and vector-memory operations.

The vector-memory commands are also sent to the *vector-memory unit* which performs the necessary cache accesses. The vector-memory unit can only send one address to the cache each cycle, but it takes advantage of the structured access patterns to load or store multiple elements with each access. The vector-memory unit communicates load and store data to and from cluster 0 in each lane to reuse the buffering already provided for the decoupled VP loads and stores.

3.4 Prototype

We are currently designing a prototype SCALE processor, and an initial floorplan is shown in Figure 13. The prototype contains a single-issue MIPS scalar control processor, four 32-bit lanes with four execution clusters each, and a 32 KB shared primary cache. The VTU has 32 registers per cluster and supports up to 128 virtual processors. The prototype's unified L1 cache is 32-way set-associative [15] and divided into four banks. The vector memory unit can perform a single access per cycle, fetching up to 128 bits from a single bank, or all lanes can perform VP accesses from different banks. The cache is non-blocking and connects to off-chip DDR2 SDRAM.

The area estimate of around 10 mm^2 in 0.18 μm technology is based on microarchitecture-level datapath designs for the control processor and VTU lanes; cell dimensions based on layout for the datapath blocks, register files, CAMs, SRAM arrays, and crossbars; and estimates for the synthesized control logic and external interface overhead. We have designed the SCALE prototype to fit into a compact area to reduce wire delays and design complexity, and to support tiling of multiple SCALE processors on a CMP for increased processing throughput. The clock frequency target is 400 MHz based on a 25 FO4 cycle time, chosen as a compromise between performance, power, and design complexity.

Vector-Thread Unit	
Number of lanes	4
Clusters per lane	4
Registers per cluster	32
AIB cache uops per cluster	32
Intra-cluster bypass latency	0 cycles
Intra-lane transport latency	1 cycle
Cross-VP transport latency	0 cycles
Clock frequency	400 MHz
L1 Unified Cache	
Size	32 KB
Associativity	32 (CAM tags)
Line size	32 B
Banks	4
Maximum bank access width	16 B
Store miss policy	write-allocate/write-back
Load-use latency	2 cycles
Memory System	
DRAM type	DDR2
Data bus width	64 bits
DRAM clock frequency	200 MHz
Data bus frequency	400 MHz
Minimum load-use latency	35 processor cycles

Table 1: Default parameters for SCALE simulations.

4. Evaluation

This section contains an evaluation and analysis of SCALE running a diverse selection of embedded benchmark codes. We first describe the simulation methodology and benchmarks, then discuss how the benchmark codes were mapped to the VT architecture and the resulting efficiency of execution.

4.1 Programming and Simulation Methodology

SCALE was designed to be compiler-friendly, and a C compiler is under development. For the results in this paper, all VTU code was hand-written in SCALE assembler (as in Figure 9) and linked with C code compiled for the MIPS control processor using `gcc`. The same binary code was used across all SCALE configurations.

A detailed cycle-level, execution-driven microarchitectural simulator has been developed based on the prototype design. Details modeled in the VTU simulation include cluster execution of micro-ops governed by execute-directives; cluster decoupling and dynamic inter-cluster data dependency resolution; memory access decoupling; operation of the vector-memory unit; operation of the command management unit, including vector-fetch and thread-fetch commands with AIB tag-checking and miss handling; and the AIB fill unit and its contention for the primary cache.

The VTU simulation is complemented by a cycle-based memory system simulation which models the multi-requester, multi-banked, non-blocking, highly-associative CAM-based cache and a detailed memory controller and DRAM model. The cache accurately models bank conflicts between different requesters; exerts back-pressure in response to cache contention; tracks pending misses and merges in new requests; and models cache-line refills and writebacks. The DRAM simulation is based on the DDR2 chips used in the prototype design, and models a 64-bit wide memory port clocked at 200 MHz (400 Mb/s/pin) including page refresh, precharge and burst effects.

The default simulation parameters are based on the prototype and are summarized in Table 1. An intra-lane transport from one cluster to another has a latency of one cycle (i.e. there will be a one cycle

bubble between the producing instruction and the dependent instruction). Cross-VP transports are able to have zero cycle latency because the clusters are physically closer together and there is less fan-in for the receive operation. Cache accesses have a two cycle latency (two bubble cycles between load and use), and accesses which miss in the cache have a minimum latency of 35 cycles.

To show scaling effects, we model four SCALE configurations with one, two, four, and eight lanes. The one, two, and four lane configurations each include four cache banks and one 64-bit DRAM port. For eight lanes, the memory system was doubled to eight cache banks and two 64-bit memory ports to appropriately match the increased compute bandwidth.

4.2 Benchmarks and Results

We have implemented a selection of benchmarks (Table 2) to illustrate the key features of SCALE, including examples from network processing, image processing, cryptography, and audio processing. The majority of these benchmarks come from the EEMBC benchmark suite. The EEMBC benchmarks may either be run "out-of-the-box" (OTB) as compiled unmodified C code, or they may be optimized (OPT) using assembly coding and arbitrary hand-tuning. This enables a direct comparison of SCALE running hand-written assembly code to optimized results from industry processors. Although OPT results match the typical way in which these processors are used, one drawback of this form of evaluation is that performance depends greatly on programmer effort, especially as EEMBC permits algorithmic and data-structure changes to many of the benchmark kernels, and optimizations used for the reported results are often unpublished. Also, not all of the EEMBC results are available for all processors, as results are often submitted for only one of the domain-specific suites (e.g., telecom).

We made algorithmic changes to several of the EEMBC benchmarks: `rotate` blocks the algorithm to enable rotating an 8-bit block completely in registers, `pktflow` implements the packet descriptor queue using an array instead of a linked list, `fir` optimizes the default algorithm to avoid copying and exploit reuse, `fbital` uses a binary search to optimize the bit allocation, `conven` uses bit packed input data to enable multiple bit-level operations to be performed in parallel, and `fft` uses a radix-2 hybrid Stockham algorithm to eliminate bit-reversal and increase vector lengths.

Figure 14 shows the simulated performance of the various SCALE processor configurations relative to several reasonable competitors from among those with the best published EEMBC benchmark scores in each domain. For each of the different benchmarks, Table 3 shows VP configuration and vector-length statistics, and Tables 4 and 5 give statistics showing the effectiveness of the SCALE control and data hierarchies. These are discussed further in the following sections.

The AMD Au1100 was included to validate the SCALE control processor OTB performance, as it has a similar structure and clock frequency, and also uses `gcc`. The Philips TriMedia TM 1300 is a five-issue VLIW processor with a 32-bit datapath. It runs at 166 MHz and has a 32 KB L1 instruction cache and 16 KB L1 data cache, with a 32-bit memory port running at 125 MHz. The Motorola PowerPC (MPC7447) is a four-issue out-of-order superscalar processor which runs at 1.3 GHz and has 32 KB separate L1 instruction and data caches, a 512 KB L2 cache, and a 64-bit memory port running at 133 MHz. The OPT results for the processor use its Altivec SIMD unit which has a 128-bit datapath and four execution units. The VIRAM processor [4] is a research vector processor with four 64-bit lanes. VIRAM runs at 200 MHz and includes 13 MB of embedded DRAM supporting up to 256 bits each of load and store data, and four independent addresses per cycle. The BOPS Manta is a clustered VLIW DSP with four clusters each capable of executing up to five instructions per cycle on 64-bit datapaths. The Manta 2.0 runs at 136 MHz with 128 KB of on-chip memory connected to a 32-bit memory port running at 136 MHz. The TI TMS TMS320C6416 is a clustered VLIW DSP with two clusters each capable of executing up to four instructions per cycle. It runs at 720 MHz and has a 16 KB instruction cache and a 16 KB data cache together with 1 MB of on-chip SRAM. The TI has a 64-bit memory interface running at 720 MHz. Apart from the Au1100 and SCALE, all other processors implement SIMD operations on packed subword values and these are widely exploited throughout the benchmark set.

Overall, the results show that SCALE can flexibly provide competitive performance with larger and more complex processors on a wide range of codes from different domains, and that performance generally scales well when adding new lanes. The results also illustrate the large speedups possible when algorithms are extensively tuned for a highly parallel processor versus compiled from standard reference code. SCALE results for `fft` and `viterbi` are not as competitive with the DSPs. This is partly due to these being preliminary versions of the code with further scope for tuning (e.g., moving the current radix-2 FFT to radix-4 and using outer-loop vectorization for `viterbi`) and partly due to the DSPs having special support for these operations (e.g., complex multiply on BOPS). We expect SCALE performance to increase significantly with the addition of subword operations and with improvements to the microarchitecture driven by these early results.

4.3 Mapping Parallelism to SCALE

The SCALE VT architecture allows software to explicitly encode the parallelism and locality available in an application. This section evaluates the architecture's expressiveness in mapping different types of code.

Data-Parallel Loops with No Control Flow

Data-parallel loops with no internal control flow, i.e. simple vectorizable loops, may be ported to the VT architecture in a similar manner as other vector architectures. Vector-fetch commands encode the cross-iteration parallelism between blocks of instructions, while vector-memory commands encode data locality and enable optimized memory access. The EEMBC image processing benchmarks (`rgbcmy`, `rgbyiq`, `hpg`) are examples of streaming vectorizable code for which SCALE is able to achieve high performance that scales with the number of lanes in the VTU. A 4-lane SCALE achieves performance competitive with VIRAM for `rgbyiq` and `rgbcmy` despite having half the main memory bandwidth, primarily because VIRAM is limited by strided accesses while SCALE refills the cache with unit-stride bursts and then has higher strided bandwidth into the cache. For the unit-stride `hpg` benchmark, performance follows memory bandwidth with the 8-lane SCALE approximately matching VIRAM.

Data-Parallel Loops with Conditionals

Traditional vector machines handle conditional code with predication (masking), but the VT architecture adds the ability to conditionally branch. Predication can be less overhead for small conditionals, but branching results in less work when conditional blocks are large. EEMBC `dither` is an example of a large conditional block in a data parallel loop. This benchmark converts a grey-scale image to black and white, and the dithering algorithm handles white pixels as a special case. In the SCALE code, each VP executes a conditional fetch for each pixel, executing only 18 SCALE instructions for white pixels versus 49 for non-white pixels.

EEMBC Benchmarks		Data Set	OTB Itr/Sec	OPT Itr/Sec	Kernel Speedup	Ops/ Cycle	Mem B/ Cycle	Loop Type DP	DC	XI	DI	DE	FT	Mem VM	VP	Description
rgbcmy	consumer	-	126	1505	11.9	6.1	3.2	×							×	RGB to CMYK color conversion
rgbyiq	consumer	-	56	1777	31.7	9.9	3.1	×							×	RGB to YIQ color conversion
hpg	consumer	-	108	3317	30.6	9.5	2.0	×						×	×	High pass grey-scale filter
text	office	-	299	435	1.5	0.3	0.0					×		×		Printer language parsing
dither	office	-	149	653	4.4	5.0	0.2	×	×					×	×	Floyd-Steinberg grey-scale dithering
rotate	office	-	704	10112	14.4	7.5	0.0	×						×	×	Binary image 90 degree rotation
lookup	network	-	1663	8850	5.3	6.3	0.0				×			×	×	IP route lookup using Patricia Trie
ospf	network	-	6346	7044	1.1	1.3	0.0						×		×	Dijkstra shortest path first
pktflow	network	512KB	6694	127677	19.1	7.8	0.6									IP packet processing
		1MB	2330	25609	11.0	3.0	3.6		×	×				×	×	
		2MB	1189	13473	11.3	3.1	3.7									
pntrch	auto	-	8771	38744	4.4	2.3	0.0						×		×	Pointer chasing, searching linked list
fir	auto	-	56724	6105006	107.6	8.7	0.3	×						×		Finite impulse response filter
fbital	telecom	typ	860	20897	24.3	4.0	0.0									Bit allocation for DSL modems
		step	12523	281938	22.5	2.5	0.0		×	×				×	×	
		pent	1304	60958	46.7	3.6	0.0									
fft	telecom	all	6572	89713	13.6	6.1	0.0	×						×		256-pt fixed-point complex FFT
viterb	telecom	all	1541	7522	4.9	4.2	0.0	×						×		Soft decision Viterbi decoder
autocor	telecom	data1	279339	3131115	11.2	4.8	0.2									Fixed-point autocorrelation
		data2	1888	64148	34.0	11.2	0.0	×						×		
		data3	1980	78751	39.8	13.0	0.0									
conven	telecom	data1	2899	2447980	844.3	9.8	0.0									Convolutional encoder
		data2	3361	3085229	917.8	10.4	0.0	×						×	×	
		data3	4259	3703703	869.4	9.5	0.1									

Other Benchmarks		Data Set	OTB Total Cycles	OPT Total Cycles	Kernel Speedup	Ops/ Cycle	Mem B/ Cycle	Loop Type DP	DC	XI	DI	DE	FT	Mem VM	VP	Description
rijndael	MiBench	large	420.8M	219.0M	2.4	2.5	0.0	×						×	×	Advanced Encryption Standard
sha	MiBench	large	141.3M	64.8M	2.2	1.8	0.0	×		×				×	×	Secure hash algorithm
qsort	MiBench	small	35.0M	21.4M	3.5	2.3	2.7						×		×	Quick sort of strings
adpcm_enc	Mediabench	-	7.7M	4.3M	1.8	2.3	0.0			×				×	×	Adaptive Differential PCM encode
adpcm_dec	Mediabench	-	6.3M	1.0M	7.9	6.7	0.0			×					×	Adaptive Differential PCM decode
li	SpecInt95	test	1,340.0M	1,151.7M	5.5	2.8	2.7	×	×	×	×			×	×	Lisp interpreter

Table 2: Benchmark Statistics and Characterization. All numbers are for the default SCALE configuration with four lanes. Results for multiple input data sets are shown separately if there was significant variation, otherwise an *all* data set indicates results were similar across inputs. As is standard practice, EEMBC statistics are for the kernel only. Total cycle numbers for non-EEMBC benchmarks are for the entire application, while the remaining statistics are for the kernel of the benchmark only (the kernel excludes benchmark overhead code and for `li` the kernel consists of the garbage collector only). The *Mem B/Cycle* column shows the DRAM bandwidth in bytes per cycle. The *Loop Type* column indicates the categories of loops which were parallelized when mapping the benchmark to SCALE: [DP] data-parallel loop with no control flow, [DC] data-parallel loop with conditional thread-fetches, [XI] loop with cross-iteration dependencies, [DI] data-parallel loop with inner-loop, [DE] loop with data-dependent exit condition, and [FT] free-running threads. The *Mem* column indicates the types of memory accesses performed: [VM] for vector-memory accesses and [VP] for individual VP loads and stores.

	rgbcmy	rgbyiq	hpg	text	dither	rotate	lookup	ospf	pktflw	pntrch	fir	fbital	fft	viterb	autcor	conven	rijnd	sha	qsort	adpcm.e	adpcm.d	li.gc	**avg.**
VP config: private regs	2.0	1.0	5.0	2.7	10.0	16.0	8.0	1.0	5.0	7.0	4.0	3.0	9.0	3.6	3.0	6.0	13.0	1.0	26.0	4.0	1.0	4.4	6.2
VP config: shared regs	10.0	18.0	3.0	3.6	16.0	3.0	9.0	5.0	12.0	14.5	2.0	8.0	1.0	3.9	2.0	7.0	5.0	3.8	20.0	19.0	17.0	5.1	8.5
vlmax	52.0	120.0	60.0	90.8	28.1	24.0	40.0	108.0	56.0	40.0	64.0	116.0	36.0	49.8	124.0	40.0	28.0	113.5	12.0	48.0	96.0	112.7	66.3
vl	52.0	120.0	53.0	6.7	24.4	18.5	40.0	1.0	52.2	12.0	60.0	100.0	25.6	16.6	32.0	31.7	4.0	5.5	12.0	47.6	90.9	62.7	39.5

Table 3: VP configuration and vector-length statistics as averages of data recorded at each vector-fetch command. The VP configuration register counts represent totals across all four clusters, `vlmax` indicates the average maximum vector length, and `vl` indicates the average vector length.

Loops with Cross-Iteration Dependencies

Many loops are non-vectorizable because they contain loop-carried data dependencies from one iteration to the next. Nevertheless, there may be ample loop parallelism available when there are operations in the loop which are not on the critical path of the cross-iteration dependency. The vector-thread architecture allows the parallelism to be exposed by making the cross-iteration (cross-VP) data transfers explicit. In contrast to software pipelining for a VLIW architecture, the vector-thread code need only schedule instructions locally in one loop iteration. As the code executes on a vector-thread machine, the dependencies between iterations resolve dynamically and the performance automatically adapts to the software critical path and the available hardware resources.

Mediabench ADPCM contains one such loop (similar to Figure 8) with two loop-carried dependencies that can propagate in parallel. The loop is mapped to a single SCALE AIB with 35 VP instructions. Cross-iteration dependencies limit the initiation interval to 5 cycles, yielding a maximum SCALE IPC of $35/5 = 7$. SCALE sustains an average of 6.7 compute-ops per cycle and achieves a speedup of $7.9\times$ compared to the control processor.

The two MiBench cryptographic kernels, `sha` and `rijndael`, have many loop-carried dependences. The `sha` mapping uses 5 cross-VP data transfers, while the `rijndael` mapping vectorizes a short four-iteration inner loop. SCALE is able to exploit instruction-level parallelism within each iteration of these kernels by using multiple clusters, but, as shown in Figure 14, performance also improves as more lanes are added.

Data-Parallel Loops with Inner-Loops

Often an inner loop has little or no available parallelism, but the outer loop iterations can run concurrently. For example, the EEMBC `lookup` code models a router using a Patricia Trie to perform IP Route Lookup. The benchmark searches the trie for each IP address in an input vector, with each lookup chasing pointers through around 5–12 nodes of the trie. Very little parallelism is

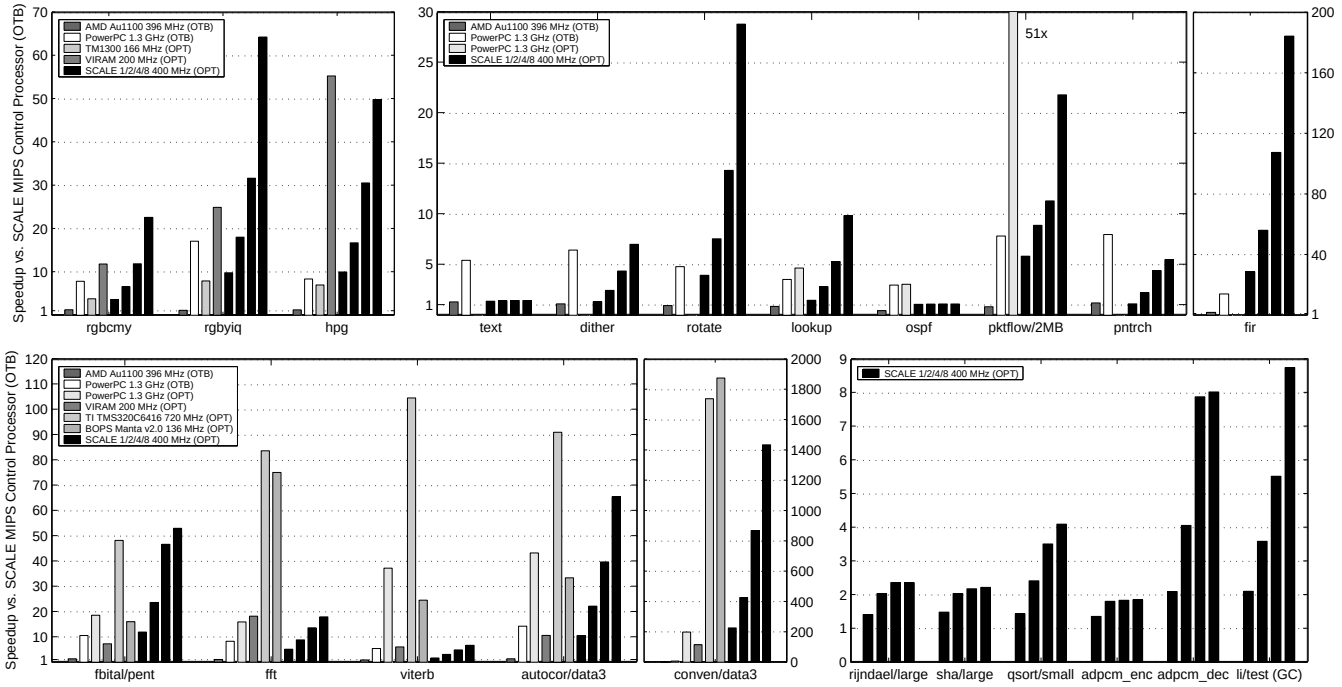

Figure 14: Performance Results: Twenty-two benchmarks illustrate the performance of four SCALE configurations (1 Lane, 2 Lanes, 4 Lanes, 8 Lanes) compared to various industry architectures. Speedup is relative to the SCALE MIPS control processor. The EEMBC benchmarks are compared in terms of iterations per second, while the non-EEMBC benchmarks are compared in terms of cycles to complete the benchmark kernel. These numbers correspond to the *Kernel Speedup* column in Table 2. For benchmarks with multiple input data sets, results for a single representative data set are shown with the data set name indicated after a forward slash.

available in each lookup, but many lookups can run simultaneously.

In the SCALE implementation, each VP handles one IP lookup using thread-fetches to traverse the trie. The ample thread parallelism keeps the lanes busy executing 6.3 ops/cycle by interleaving the execution of multiple VPs to hide memory latency. Vector-fetches provide an advantage over a pure multithreaded machine by efficiently distributing work to the VPs, avoiding contention for a shared work queue. Additionally, vector-load commands optimize the loading of IP addresses before the VP threads are launched.

Reductions and Data-Dependent Loop Exit Conditions

SCALE provides efficient support for arbitrary reduction operations by using shared registers to accumulate partial reduction results from multiple VPs on each lane. The shared registers are then combined across all lanes at the end of the loop using the cross-VP network. The `pktflow` code uses reductions to count the number of packets processed.

Loops with data-dependent exit conditions ("while" loops) are difficult to parallelize because the number of iterations is not known in advance. For example, the `strcmp` and `strcpy` standard C library routines used in the `text` benchmark loop until the string termination character is seen. The cross-VP network can be used to communicate exit status across VPs but this serializes execution. Alternatively, iterations can be executed speculatively in parallel and then nullified after the correct exit iteration is determined. The check to find the exit condition is coded as a cross-iteration reduction operation. The `text` benchmark executes most of its code on the control processor, but uses this technique for the string routines to attain a 1.5× overall speedup.

Free-Running Threads

When structured loop parallelism is not available, VPs can be used to exploit arbitrary thread parallelism. With free-running threads, the control processor interaction is eliminated. Each VP thread runs in a continuous loop getting tasks from a work-queue accessed using atomic memory operations. An advantage of this method is that it achieves good load-balancing between the VPs and can keep the VTU constantly utilized.

Three benchmarks were mapped with free-running threads. The `pntrch` benchmark searches for tokens in a doubly-linked list, and allows up to five searches to execute in parallel. The `qsort` benchmark uses quick-sort to alphabetize a list of words. The SCALE mapping recursively divides the input set and assigns VP threads to sort partitions, using VP function calls to implement the compare routine. The benchmark achieves 2.3 ops/cycle despite a high cache miss rate. The `ospf` benchmark has little available parallelism and the SCALE implementation maps the code to a single VP to exploit ILP for a small speedup.

Mixed Parallelism

Some codes exploit a mixture of parallelism types to accelerate performance and improve efficiency. The garbage collection portion of the lisp interpreter (`li`) is split into two phases: mark, which traverses a tree of currently live lisp nodes and sets a flag bit in every visited node, and sweep, which scans through the array of nodes and returns a linked list containing all of the unmarked nodes. During mark, the SCALE code sets up a queue of nodes to be processed and uses a stripmine loop to examine the nodes, mark them, and enqueue their children. In the sweep phase, VPs are assigned segments of the allocation array and then each construct a list of unmarked nodes within their segment in parallel. Once the VP threads terminate, the control processor vector-fetches an AIB that stitches the individual lists together using cross-VP data transfers, thus producing the intended structure. Although the garbage collector has a high cache miss rate, the high degree of parallelism exposed in this way allows SCALE to sustain 2.8 operations/cycle and attain a

	rgbcmy	rgbyiq	hpg	text	dither	rotate	lookup	ospf	pktflw	pntrch	fir	fbital	fft	viterb	autcor	conven	rijnd	sha	qsort	adpcm.e	adpcm.d	li.gc
compute-ops / AIB	21.0	29.0	3.7	4.9	8.6	19.7	5.1	16.5	4.2	7.0	4.0	7.4	3.0	8.8	3.0	7.3	13.4	14.1	9.1	61.0	35.0	8.9
compute-ops / AIB tag-check	273.0	870.0	48.6	8.2	10.4	91.1	5.3	18.5	21.5	7.0	59.6	14.2	19.4	36.8	24.0	57.5	13.4	25.4	9.1	726.2	795.3	12.2
compute-ops / ctrl. proc. instr.	136.0	431.9	44.7	0.2	23.7	85.7	639.2	857.8	152.3	3189.6	18.1	62.8	5.8	4.9	23.6	19.7	8.9	3.9	5.8	229.2	186.0	122.7
thread-fetches / VP thread	0.0	0.0	0.0	0.0	3.8	0.0	26.7	3969.0	0.2	3023.7	0.0	1.0	0.0	0.0	0.0	0.0	0.9	0.0	113597.9	0.0	0.0	2.4
AIB cache miss percent	0.0	0.0	0.0	0.0	0.0	33.2	0.0	22.5	0.0	1.5	1.6	0.0	0.2	0.0	0.0	0.0	0.0	0.0	4.3	0.0	0.1	0.4

Table 4: Control hierarchy statistics. The first three rows show are the average number of compute-ops per executed AIB, per AIB tag-check (caused by either a vector-fetch, VP-fetch, or thread-fetch), and per executed control processor instruction. The next row shows the average number thread-fetches issued by each dynamic VP thread (launched by a vector-fetch or VP-fetch). The last row shows the miss rate for AIB tag-checks.

	rgbcmy	rgbyiq	hpg	text	dither	rotate	lookup	ospf	pktflw	pntrch	fir	fbital	fft	viterb	autcor	conven	rijnd	sha	qsort	adpcm.e	adpcm.d	li.gc	**avg.**
sources: chain register	75.6	92.9	40.0	31.2	41.3	5.8	21.0	13.1	62.7	31.0	38.8	30.5	31.9	37.1	48.4	46.8	81.5	115.8	32.6	20.3	34.1	39.4	44.2
sources: register file	99.3	86.0	106.7	75.3	94.2	109.8	113.6	127.0	84.7	115.0	113.3	114.4	84.5	87.3	96.9	90.6	72.1	27.9	102.0	97.4	110.1	77.6	94.8
sources: immediate	28.4	31.0	6.7	13.1	27.2	64.0	21.8	52.9	45.7	38.8	2.6	30.2	7.5	13.9	0.0	50.0	23.1	38.9	66.6	35.4	38.7	71.7	32.2
dests: chain register	56.7	58.5	40.0	18.2	43.8	5.8	21.8	18.5	77.9	38.5	38.8	22.8	31.9	37.1	48.4	40.6	81.1	84.5	32.9	12.8	24.8	39.2	39.8
dests: register file	33.8	31.2	60.0	52.8	46.0	59.6	48.2	87.3	52.6	23.1	60.7	83.0	51.2	64.9	51.5	75.0	22.0	15.5	43.1	81.8	44.2	26.8	50.6
ext. cluster transports	52.0	51.6	53.3	43.0	45.9	34.5	36.6	53.5	57.6	30.9	38.8	90.7	31.9	74.1	48.5	40.6	29.5	68.3	13.9	72.7	21.7	56.3	47.5
load elements	14.2	10.3	20.0	14.6	14.7	5.7	14.8	21.8	22.0	15.4	19.0	15.1	20.7	13.9	25.0	12.5	28.4	15.4	18.1	6.4	9.3	13.0	15.9
load addresses	14.2	10.3	5.3	3.7	8.1	1.7	14.2	21.8	20.8	15.4	5.4	9.4	8.0	5.3	7.4	7.9	25.7	12.3	18.1	4.8	9.3	11.6	10.9
load bytes	14.2	10.3	20.0	14.6	30.2	5.7	59.1	87.4	54.2	61.6	75.9	30.2	41.3	38.4	49.9	25.1	113.4	61.7	64.9	21.4	27.9	52.0	43.6
load bytes from DRAM	14.2	10.3	7.5	0.0	2.9	0.3	0.2	0.0	115.3	0.0	0.4	0.0	0.0	0.0	0.1	0.0	0.0	0.0	59.1	0.0	0.0	39.2	11.3
store elements	4.7	10.3	6.7	4.8	3.5	5.8	0.0	10.4	1.7	0.3	1.0	0.5	16.9	9.3	0.0	3.1	3.3	3.5	15.5	1.1	3.1	9.5	5.2
store addresses	4.7	10.3	1.8	1.4	1.8	5.8	0.0	10.4	0.4	0.3	0.5	0.1	4.2	6.8	0.0	0.8	1.6	3.0	15.5	1.1	3.1	9.5	3.8
store bytes	18.9	10.3	6.7	4.8	6.0	5.8	0.0	41.5	6.8	1.2	4.2	1.0	33.8	29.1	0.1	12.5	13.2	14.0	62.1	1.1	6.2	38.1	14.4
store bytes to DRAM	18.9	10.3	6.7	0.5	0.7	0.0	0.0	0.0	8.4	0.0	0.1	0.0	0.0	0.0	0.0	0.0	0.0	0.0	50.5	0.0	0.0	46.6	6.5

Table 5: Data hierarchy statistics. The counts are scaled to reflect averages per 100 compute-ops executed in each benchmark, and the average (avg) column gives equal weight to all the benchmarks. Compute-op sources are broken down as coming from chain registers, the register file, or immediates; and compute-op and writeback-op destinations are broken down as targeting chain registers or the register file. The *ext. cluster transports* row reflects the number of results sent to external clusters. The *load elements* row reflects the number of elements accessed by either VP loads or vector-loads, while the *load addresses* row reflects the number of cache accesses. The *load bytes* row reflects the total number of bytes for the VP loads and vector-loads, while the *load bytes from DRAM* row reflects the DRAM bandwidth used to retrieve this data. The breakdown for stores corresponds to the breakdown for loads.

speedup of 5.5 over the control processor alone.

4.4 Locality and Efficiency

The strength of the SCALE VT architecture is its ability to capture a wide variety of parallelism in applications while using simple microarchitectural mechanisms that exploit locality in both control and data hierarchies.

A VT machine amortizes control overhead by exploiting the locality exposed by AIBs and vector-fetch commands, and by factoring out common control code to run on the control processor. A vector-fetch broadcasts an AIB address to all lanes and each lane performs a single tag-check to determine if the AIB is cached. On a hit, an execute directive is sent to the clusters which then retrieve the instructions within the AIB using a short (5-bit) index into the small AIB cache. The cost of each instruction fetch is on par with a register file read. On an AIB miss, a vector-fetch will broadcast AIBs to refill all lanes simultaneously. The vector-fetch ensures an AIB will be reused by each VP in a lane before any eviction is possible. When an AIB contains only a single instruction on a cluster, a vector-fetch will keep the ALU control lines fixed while each VP executes its operation, further reducing control energy.

As an example of amortizing control overhead, `rbgyiq` runs on SCALE with a vector-length of 120 and vector-fetches an AIB with 29 VP instructions. Thus, each vector-fetch executes 3,480 instructions on the VTU, 870 instructions per tag-check in each lane. This is an extreme example, but vector-fetches commonly execute 10s–100s of instructions per tag-check even for non-vectorizable loops such as `adpcm` (Table 4).

AIBs also help in the data hierarchy by allowing the use of chain registers, which reduces register file energy; and sharing of temporary registers, which reduces the register file size needed for a large number of VPs. Table 5 shows that chain registers comprise around 32% of all register sources and 44% of all register destinations. Table 3 shows that across all benchmarks, VP configurations use an average of 8.5 shared and 6.2 private registers, with an average maximum vector length above 64 (16 VPs per lane). The significant variability in register requirements for different kernels stresses the importance of allowing software to configure VPs with just enough of each register type.

Vector-memory commands enforce spatial locality by moving data between memory and the VP registers in groups. This improves performance and saves memory system energy by avoiding the additional arbitration, tag-checks, and bank conflicts that would occur if each VP requested elements individually. Table 5 shows the reduction in memory addresses from vector-memory commands. The maximum improvement is a factor of four, when each vector cache access loads or stores one element per lane. The VT architecture can exploit memory data-parallelism even in loops with non-data-parallel compute. For example, the `fbital`, `text`, and `adpcm_enc` benchmarks use vector-memory commands to access data for vector-fetched AIBs with cross-VP dependencies.

Table 5 shows that the SCALE data cache is effective at reducing DRAM bandwidth for most of the benchmarks. Two exceptions are the `pktflow` and `li` benchmarks for which the DRAM bytes transferred exceed the total bytes accessed. The current design always transfers 32-byte lines on misses, but support for non-allocating loads and stores could help reduce the bandwidth for these benchmarks.

Clustering in SCALE is area and energy efficient and cluster decoupling improves performance. The clusters each contain only a subset of all possible functional units and a small register file with few ports, reducing size and wiring energy. Each cluster executes compute-ops and inter-cluster transport operations in order, requiring only simple interlock logic with no inter-thread arbitra-

tion or dynamic inter-cluster bypass detection. Independent control on each cluster enables decoupled cluster execution to hide large inter-cluster or memory latencies. This provides a very cheap form of SMT where each cluster can be executing code for different VPs on the same cycle (Figure 12).

5. Related Work

The VT architecture draws from earlier vector architectures [9], and like vector microprocessors [14, 6, 3] the SCALE VT implementation provides high throughput at low complexity. Similar to CODE [5], SCALE uses decoupled clusters to simplify chaining control and to reduce the cost of a large vector register file supporting many functional units. However, whereas CODE uses register renaming to hide clusters from software, SCALE reduces hardware complexity by exposing clustering and statically partitioning inter-cluster transport and writeback operations.

The Imagine [8] stream processor is similar to vector machines, with the main enhancement being the addition of stream load and store instructions that pack and unpack arrays of multi-field records stored in DRAM into multiple vector registers, one per field. In comparison, SCALE uses a conventional cache to enable unit-stride transfers from DRAM, and provides segment vector-memory commands to transfer arrays of multi-field records between the cache and VP registers. Like SCALE, Imagine improves register file locality compared with traditional vector machines by executing all operations for one loop iteration before moving to the next. However, Imagine instructions use a low-level VLIW ISA that exposes machine details such as number of physical registers and lanes, whereas SCALE provides a higher-level abstraction based on VPs and AIBs.

VT enhances the traditional vector model to support loops with cross-iteration dependencies and arbitrary internal control flow. Chiueh's multi-threaded vectorization [1] extends a vector machine to handle loop-carried dependencies, but is limited to a single lane and requires the compiler to have detailed knowledge of all functional unit latencies. Jesshope's micro-threading [2] uses a vector-fetch to launch micro-threads which each execute one loop iteration, but whose execution is dynamically scheduled on a per-instruction basis. In contrast to VT's low-overhead direct cross-VP data transfers, cross-iteration synchronization is done using full/empty bits on shared global registers. Like VT, Multiscalar [12] statically determines loop-carried register dependencies and uses a ring to pass cross-iteration values. But Multiscalar uses speculative execution with dynamic checks for memory dependencies, while VT dispatches multiple non-speculative iterations simultaneously. Multiscalar can execute a wider range of loops in parallel, but VT can execute many common parallel loop types with much simpler logic and while using vector-memory operations.

Several other projects are developing processors capable of exploiting multiple forms of application parallelism. The Raw [13] project connects a tiled array of simple processors. In contrast to SCALE's direct inter-cluster data transfers and cluster decoupling, inter-tile communication on Raw is controlled by programmed switch processors and must be statically scheduled to tolerate latencies. The Smart Memories [7] project has developed an architecture with configurable processing tiles which support different types of parallelism, but it has different instruction sets for each type and requires a reconfiguration step to switch modes. The TRIPS processor [10] similarly must explicitly *morph* between instruction, thread, and data parallelism modes. These mode switches limit the ability to exploit multiple forms of parallelism at a fine-grain, in contrast to SCALE which seamlessly combines vector and threaded execution while also exploiting local instruction-level parallelism.

6. Conclusion

The vector-thread architectural paradigm allows software to more efficiently encode the parallelism and locality present in many applications, while the structure provided in the hardware-software interface enables high-performance implementations that are efficient in area and power. The VT architecture unifies support for all types of parallelism and this flexibility enables new ways of parallelizing codes, for example, by allowing vector-memory operations to feed directly into threaded code. The SCALE prototype demonstrates that the VT paradigm is well-suited to embedded applications, allowing a single relatively small design to provide competitive performance across a range of application domains. Although this paper has focused on applying VT to the embedded domain, we anticipate that the vector-thread model will be widely applicable in other domains including scientific computing, high-performance graphics processing, and machine learning.

7. Acknowledgments

This work was funded in part by DARPA PAC/C award F30602-00-2-0562, NSF CAREER award CCR-0093354, an NSF graduate fellowship, donations from Infineon Corporation, and an equipment donation from Intel Corporation.

8. References

[1] T.-C. Chiueh. Multi-threaded vectorization. In *ISCA-18*, May 1991.

[2] C. R. Jesshope. Implementing an efficient vector instruction set in a chip multi-processor using micro-threaded pipelines. *Australia Computer Science Communications*, 23(4):80–88, 2001.

[3] K. Kitagawa, S. Tagaya, Y. Hagihara, and Y. Kanoh. A hardware overview of SX-6 and SX-7 supercomputer. *NEC Research & Development Journal*, 44(1):2–7, Jan 2003.

[4] C. Kozyrakis. *Scalable vector media-processors for embedded systems*. PhD thesis, University of California at Berkeley, May 2002.

[5] C. Kozyrakis and D. Patterson. Overcoming the limitations of conventional vector processors. In *ISCA-30*, June 2003.

[6] C. Kozyrakis, S. Perissakis, D. Patterson, T. Anderson, K. Asanović, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick. Scalable Processors in the Billion-Transistor Era: IRAM. *IEEE Computer*, 30(9):75–78, Sept 1997.

[7] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. Dally, and M. Horowitz. Smart Memories: A modular reconfigurable architecture. In *Proc. ISCA 27*, pages 161–171, June 2000.

[8] S. Rixner, W. Dally, U. Kapasi, B. Khailany, A. Lopez-Lagunas, P. Mattson, and J. Owens. A bandwidth-efficient architecture for media processing. In *MICRO-31*, Nov 1998.

[9] R. M. Russel. The CRAY-1 computer system. *Communications of the ACM*, 21(1):63–72, Jan 1978.

[10] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D. Burger, S. W. Keckler, and C. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *ISCA-30*, June 2003.

[11] J. E. Smith. Dynamic instruction scheduling and the Astronautics ZS-1. *IEEE Computer*, 22(7):21–35, July 1989.

[12] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *ISCA-22*, pages 414–425, June 1995.

[13] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *IEEE Computer*, 30(9):86–93, Sept 1997.

[14] J. Wawrzynek, K. Asanović, B. Kingsbury, J. Beck, D. Johnson, and N. Morgan. Spert-II: A vector microprocessor system. *IEEE Computer*, 29(3):79–86, Mar 1996.

[15] M. Zhang and K. Asanović. Highly-associative caches for low-power processors. In *Kool Chips Workshop, MICRO-33*, Dec 2000.

Appears in: The First International Conference on Mobile Systems, Applications, and Services, San Francisco, CA, May 2003. Received Best Paper Award.

APPENDIX B - Energy Aware Lossless Data Compression

Kenneth Barr and Krste Asanović

MIT Laboratory for Computer Science
200 Technology Square, Cambridge, MA 02139
E-mail: {kbarr,krste}@lcs.mit.edu

Abstract

Wireless transmission of a bit can require over 1000 times more energy than a single 32-bit computation. It would therefore seem desirable to perform significant computation to reduce the number of bits transmitted. If the energy required to compress data is less than the energy required to send it, there is a net energy savings and consequently, a longer battery life for portable computers. This paper reports on the energy of lossless data compressors as measured on a StrongARM SA-110 system. We show that with several typical compression tools, there is a net energy *increase* when compression is applied before transmission. Reasons for this increase are explained, and hardware-aware programming optimizations are demonstrated. When applied to Unix *compress*, these optimizations improve energy efficiency by 51%. We also explore the fact that, for many usage models, compression and decompression need not be performed by the same algorithm. By choosing the lowest-energy compressor and decompressor on the test platform, rather than using default levels of compression, overall energy to send compressible web data can be reduced 31%. Energy to send harder-to-compress English text can be reduced 57%. Compared with a system using a single optimized application for both compression and decompression, the asymmetric scheme saves 11% or 12% of the total energy depending on the dataset.

1 Introduction

Wireless communication is an essential component of mobile computing, but the energy required for transmission of a single bit has been measured to be over 1000 times greater than a single 32-bit computation. Thus, if 1000 computation operations can compress data by even one bit, energy should be saved. However, accessing memory can be over 200 times more costly than computation on our test platform, and it is memory access that dominates most lossless data compression algorithms. In fact, even moderate compression (e.g. `gzip -6`) can require so many memory accesses that one observes an *increase* in the overall energy required to send certain data.

While some types of data (e.g., audio and video) may accept some degradation in quality, other data must be transmitted faithfully with no loss of information. Fidelity can not be sacrificed to reduce energy as is done in related work on lossy compression. Fortunately, an understanding of a program's behavior and the energy required by major hardware components can be used to reduce energy. The ability to efficiently perform efficient lossless compression also provides second-order benefits such as reduction in packet loss and less contention for the fixed wireless bandwidth. Concretely, if n bits have been compressed to m bits ($n > m$); c is the cost of compression and decompression; and w is the cost per bit of transmission and reception; compression is energy efficient if $\frac{c}{n-m} < w$. This paper examines the elements of this inequality and their relationships.

We measure the energy requirements of several lossless data compression schemes using the "Skiff" platform developed by Compaq Cambridge Research Labs. The Skiff is a StrongARM-based system designed with energy measurement in mind. Energy usage for CPU, memory, network card, and peripherals can be measured individually. The platform is similar to the popular Compaq iPAQ handheld computer, so the results are relevant to handheld hardware and developers of embedded software. Several families of compression algorithms are analyzed and characterized, and it is shown that carelessly applying compression prior to transmission may cause an overall energy increase. Behaviors and resource-usage patterns are highlighted which allow for energy-efficient lossless compression of data by applications or network drivers. We focus on situations in which the mixture of high energy network operations and low energy processor operations can be adjusted so that overall energy is lower. This is possible even if the number of total opera-

tions, or time to complete them, increases. Finally, a new energy-aware data compression strategy composed of an asymmetric compressor and decompressor is presented and measured.

Section 2 describes the experimental setup including equipment, workloads, and the choice of compression applications. Section 3 begins with the measurement of an encouraging communication-computation gap, but shows that modern compression tools do not exploit the the low relative energy of computation versus communication. Factors which limit energy reduction are presented. Section 4 applies an understanding of these factors to reduce overall energy of transmission though hardware-conscious optimizations and asymmetric compression choices. Section 5 discusses related work, and Section 6 concludes.

2 Experimental setup

While simulators may be tuned to provide reasonably accurate estimations of a particular system's energy, observing real hardware ensures that complex interactions of components are not overlooked or oversimplified. This section gives a brief description of our hardware and software platform, the measurement methodology, and benchmarks.

2.1 Equipment

The Compaq Personal Server, codenamed "Skiff," is essentially an initial, "spread-out" version of the Compaq iPAQ built for research purposes [13]. Powered by a 233 MHz StrongARM SA-110 [29, 17], the Skiff is computationally similar to the popular Compaq iPAQ handheld (an SA-1110 [18] based device). For wireless networking, we add a five volt Enterasys 802.11b wireless network card (part number CSIBD-AA). The Skiff has 32 MB of DRAM, support for the Universal Serial Bus, a RS232 Serial Port, Ethernet, two Cardbus sockets, and a variety of general purpose I/O. The Skiff PCB boasts separate power planes for its CPU, memory and memory controller, and other peripherals allowing each to be measured in isolation (Figure 1). With a Cardbus extender card, one can isolate the power used by a wireless network card as well. A programmable multimeter and sense resistor provide a convenient way to examine energy in a active system with error less than 5% [47].

The Skiff runs ARM/Linux 2.4.2-rmk1-np1-hh2 with PCMCIA Card Services 3.1.24. The Skiff has only 4 MB of non-volatile flash memory to contain a file system, so the root filesystem is mounted via NFS using the wired ethernet port. For benchmarks which require file system access, the executable and input dataset is brought into RAM before timing begins. This is verified by observing

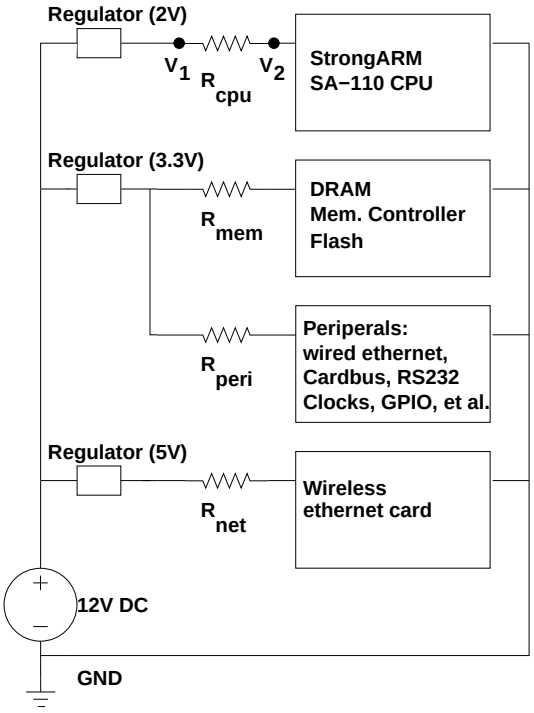

Figure 1. Simplified Skiff power schematic

the cessation of traffic on the network once the program completes loading. I/O is conducted in memory using a modified SPEC harness [42] to avoid the large cost of accessing the network filesystem.

2.2 Benchmarks

Figure 2 shows the performance of several lossless data compression applications using metrics of compression ratio, execution time, and static memory allocation. The datasets are the first megabyte (English books and a bibliography) from the Calgary Corpus [5] and one megabyte of easily compressible web data (mostly HTML, Javascript, and CSS) obtained from the homepages of the Internet's most popular websites [32, 25]. Graphics were omitted as they are usually in compressed form already and can be recognized by application-layer software via their file extensions. Most popular repositories ([4, 10, 11]) for comparison of data compression do not examine the memory footprint required for compression or decompression. Though static memory usage may not always reflect the size of the application's working set, it is an essential consideration in mobile computing where memory is a more precious resource. A detailed look at the memory used by each application, and its effect on time, compression ratio, and energy will be presented in Section 3.3.

Figure 2 confirms that we have chosen an array of ap-

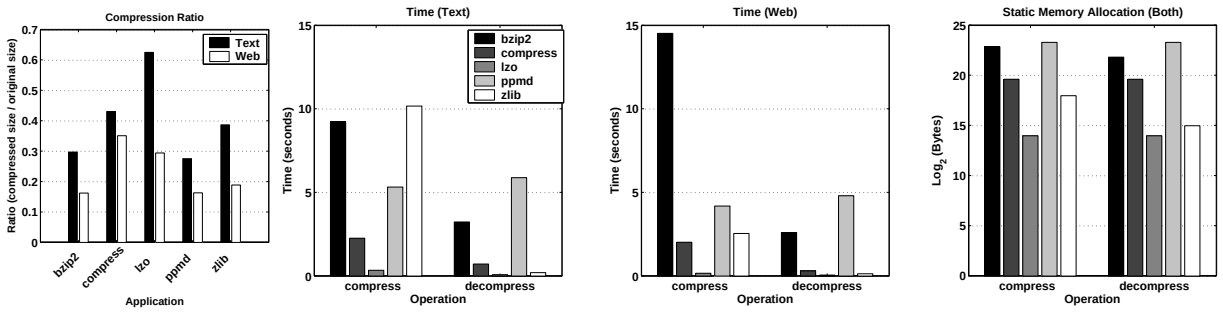

Figure 2. Benchmark comparison by traditional metrics

plications that span a range of compression ratios and execution times. Each application represents a different family of compression algorithms as noted in Table 1. Consideration was also given to popularity and documentation, as well as quality, parameterizability, and portability of the source code. The table includes the default parameters used with each program. To avoid unduly handicapping any algorithm, it is important to work with well-implemented code. Mature applications such as *compress*, *bzip2*, and *zlib* reflect a series of optimizations that have been applied since their introduction. While *PPMd* is an experimental program, it is effectively an optimization of the Prediction by Partial Match (PPM) compressors that came before it. *LZO* represents an approach for achieving great speed with LZ77. Each of the five applications is summarized below assuming some familiarity with each algorithm. A more complete treatment with citations may be found in [36].

zlib combines LZ77 and Huffman coding to form an algorithm known as "deflate." The LZ77 sliding window size and hash table memory size may be set by the user. LZ77 tries to replace a string of symbols with a pointer to the longest prefix match previously encountered. A larger window improves the ability to find such a match. More memory allows for less collisions in the *zlib* hash table. Users may also set an "effort" parameter which dictates how hard the compressor should try to extend matches it finds in its history buffer. *zlib* is the library form of the popular *gzip* utility (the library form was chosen as it provides more options for trading off memory and performance). Unless specified, it is configured with parameters similar to *gzip*.

LZO is a compression library meant for "real-time" compression. Like *zlib*, it uses LZ77 with a hash table to perform searches. *LZO* is unique in that its hash table can be sized to fit in 16KB of memory so it can remain in cache. Its small footprint, coding style (it is written completely with macros to avoid function call overhead), and ability to read and write data "in-place" without additional copies make *LZO* extremely fast. In the interest of speed, its hash table can only store pointers to 4096 matches, and no effort is made to find the longest match. Match length and offset are encoded more simply than in *zlib*.

compress is a popular Unix utility. It implements the LZW algorithm with codewords beginning at nine bits. Though a bit is wasted for each single 8-bit character, once longer strings have been seen, they may be replaced with short codes. When all nine-bit codes have been used, the codebook size is doubled and the use of ten-bit codes begins. This doubling continues until codes are sixteen bits long. The dictionary becomes static once it is entirely full. Whenever *compress* detects decreasing compression ratio, the dictionary is cleared and the process beings anew. Dictionary entries are stored in a hash table. Hashing allows an average constant-time access to any entry, but has the disadvantage of poor spatial locality when combining multiple entries to form a string. Despite the random dispersal of codes to the table, common strings may benefit from temporal locality. To reduce collisions, the table should be sparsely filled which results in wasted memory. During decompression, each table entry may be inserted without collision.

PPMd is a recent implementation of the PPM algorithm. Windows users may unknowingly be using *PPMd* as it is the text compression engine in the popular *WinRAR* program. PPM takes advantage of the fact that the occurrence of a certain symbol can be highly dependent on its context (the string of symbols which preceded it). The PPM scheme maintains such context information to estimate the probability of the next input symbol to appear. An arithmetic coder uses this stream of probabilities to efficiently code the source. As the model becomes more accurate, the occurrence of a highly likely symbol requires fewer bits to encode. Clearly, longer contexts will improve the probability estimation, but it requires time to amass large contexts (this is similar to the startup effect in LZ78). To account for this, "escape symbols" exist to progressively step down to shorter context lengths. This introduces a trade-off in which encoding a long series of escape symbols can require more space than is saved by the use of large contexts. Stor-

Application (Version)	Source	Algorithm	Notes (defaults)
bzip2 (0.1pl2)	[37]	BWT	RLE→BWT→MTF→RLE→HUFF (900k block size)
compress (4.0)	[21]	LZW	modified Unix Compress based on Spec95 (16 bit codes (maximum))
LZO (1.07)	[33]	LZ77	Favors speed over compression (lzo1x_12. 4K entry hash table uses 16KB)
PPMd (variant I)	[40]	PPM	used in "rar" compressor (Order 4, 10MB memory, restart model)
zlib (1.1.4)	[9]	LZ77	library form of gzip (Chaining level 6 / 32K Window / 32K Hash Table)

Table 1. Compression applications and their algorithms

ing and searching through each context accounts for the large memory requirements of PPM schemes. The length of the maximum context can be varied by *PPMd*, but defaults to four. When the context tree fills up, *PPMd* can clear and start from scratch, freeze the model and continue statically, or prune sections of the tree until the model fits into memory.

bzip2 is based on the Burrows Wheeler Transform (BWT) [8]. The BWT converts a block S of length n into a pair consisting of a permutation of S (call it L) and an integer in the interval $[0..n-1]$. More important than the details of the transformation is its effect. The transform collects groups of identical input symbols such that the probability of finding a symbol s in a region of L is very high if another instance of s is nearby. Such an L can be processed with a "move-to-front" coder which will yield a series consisting of a small alphabet: runs of zeros punctuated with low numbers which in turn can be processed with a Huffman or Arithmetic coder. For processing efficiency, long runs can be filtered with a run length encoder. As block size is increased, compression ratio improves. Diminishing returns (with English text) do not occur until block size reaches several tens of megabytes. Unlike the other algorithms, one could consider BWT to take advantage of symbols which appear in the *"future"*, not just those that have passed. *bzip2* reads in blocks of data, run-length-encoding them to improve sort speed. It then applies the BWT and uses a variant of move-to-front coding to produce a compressible stream. Though the alphabet may be large, codes are only created for symbols in use. This stream is run-length encoded to remove any long runs of zeros. Finally Huffman encoding is applied. To speed sorting, *bzip2* applies a modified quicksort which has memory requirements over five times the size of the block.

2.3 Performance and implementation concerns

A compression algorithm may be implemented with many different, yet reasonable, data structures (including binary tree, splay tree, trie, hash table, and list) and yield vastly different performance results [3]. The quality and applicability of the implementation is as important as the underlying algorithm. This section has presented implementations from each algorithmic family. By choosing a top representative in each family, the implementation playing field is leveled, making it easier to gain insight into the underlying algorithm and its influence on energy. Nevertheless, it is likely that each application can be optimized further (Section 4.1 shows the benefit of optimization) or use a more uniform style of I/O. Thus, evaluation must focus on inherent patterns rather than making a direct quantitative comparison.

3 Observed Energy of Communication, Computation, and Compression

In this section, we observe that over 1000 32 bit ADD instructions can be executed by the Skiff with the same amount of energy it requires to send a single bit via wireless ethernet. This fact motivates the investigation of pre-transmission compression of data to reduce overall energy. Initial experiments reveal that reducing the number of bits to send does not always reduce the total energy of the task. This section elaborates on both of these points which necessitate the in-depth experiments of Section 3.3.

3.1 Raw Communication-to-Computation Energy Ratio

To quantify the gap between wireless communication and computation, we have measured wireless idle, send, and receive energies on the Skiff platform. To eliminate competition for wireless bandwidth from other devices in the lab, we established a dedicated channel and ran the network in ad-hoc mode consisting of only two wireless nodes. We streamed UDP packets from one node to the other; UDP was used to eliminate the effects of waiting for an ACK. This also insures that receive tests measure only receive energy and send tests measure only send energy. This setup is intended to find the minimum network energy by removing arbitration delay and the energy of TCP overhead to avoid biasing our results.

With the measured energy of the transmission and the size of data file, the energy required to send or receive a bit can be derived. The results of these network benchmarks appear in Figure 3 and are consistent with other studies [20]. The card is set to its maximum speed of

11 Mb/s and two tests are conducted. In the first, the Skiff communicates with a wireless card mere inches away and achieves 5.70 Mb/sec. In the second, the second node is placed as far from the Skiff as possible without losing packets. Only 2.85 Mb/sec is achieved. These two cases bound the performance of our 11 Mb/sec wireless card; typical performance should be somewhere between them.

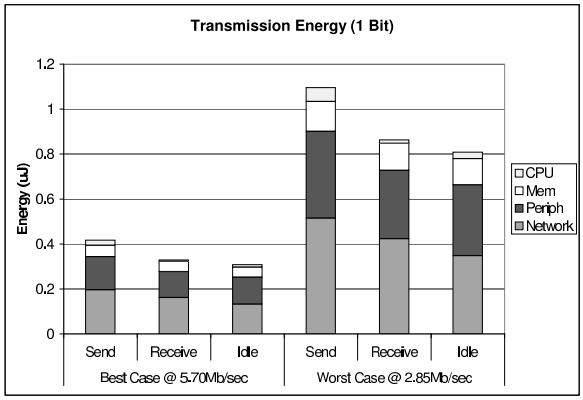

Figure 3. Measured communication energy of Enterasys wireless NIC

Next, a microbenchmark is used to determine the minimum energy for an ADD instruction. We use Linux boot code to bootstrap the processor; select a cache configuration; and launch assembly code unencumbered by an operating system. One thousand ADD instructions are followed by an unconditional branch which repeats them. This code was chosen and written in assembly language to minimize effects of the branch. Once the program has been loaded into instruction cache, the energy used by the processor for a single add is 0.86 nJ.

From these initial network and ADD measurements, we can conclude that sending a single bit is roughly equivalent to performing 485–1267 ADD operations depending on the quality of the network link ($\frac{4.17\times10^{-7}\,\mathrm{J}}{0.86\times10^{-9}\,\mathrm{J}} \approx 485$ or $\frac{1.09\times10^{-6}\,\mathrm{J}}{0.86\times10^{-9}\,\mathrm{J}} \approx 1267$). This gap of 2–3 orders of magnitude suggests that much additional effort can be spent trying to reduce a file's size before it is sent or received. But the issue is not so simple.

3.2 Application-Level Communication-to-Computation Energy Ratio

On the Skiff platform, memory, peripherals, and the network card remain powered on even when they are not active, consuming a fixed energy overhead. They may even switch when not in use in response to changes on shared buses. The energy used by these components during the ADD loop is significant and is shown in Table 2. Once a task-switching operating system is loaded and other applications vie for processing time, the communication-to-computation energy ratio will decrease further. Finally, the applications examined in this paper are more than a mere series of ADDs; the variety of instructions (especially Loads and Stores) in compression applications shrinks the ratio further.

Network card	0.43 nJ
CPU	**0.86 nJ**
Mem	1.10 nJ
Periph	4.20 nJ
Total	6.59 nJ

Table 2. Total Energy of an ADD

The first row of Figures 4 and 5 show the energy required to compress our text and web dataset and transmit it via wireless ethernet. To avoid punishing the benchmarks for the Skiff's high power, idle energy has been removed from the peripheral component so that it represents only the amount of *additional* energy (due to bus toggling and arbitration effects) over and above the energy that would have been consumed by the peripherals remaining idle for the duration of the application. Idle energy is not removed from the memory and CPU portions as they are required to be active for the duration of the application. The network is assumed to consume no power until it is turned on to send or receive data. The popular compression applications discussed in Section 2.2 are used with their default parameters, and the rightmost bar shows the energy of merely copying the uncompressed data over the network. Along with energy due to default operation (labeled "bzip2-900," "compress-16," "lzo-16," "ppmd-10240," and "zlib-6"), the figures include energy for several invocations of each application with varying parameters. *bzip2* is run with both the default 900 KB block sizes as well as its smallest 100 KB block. *compress* is also run at both ends of its spectrum (12 bit and 16 bit maximum codeword size). *LZO* runs in just 16 KB of working memory. *PPMd* uses 10 MB, 1 MB, and 32 KB memory with the cutoff mechanism for freeing space (as it is faster than the default "restart" in low-memory configurations). *zlib* is run in a configuration similar to *gzip*. The numeric suffix (9, 6, or 1) refers to effort level and is analogous to *gzip*'s commandline option. These various invocations will be studied in section 3.3.3.

While most compressors do well with the web data, in several cases the energy to compress the file approaches or outweighs the energy to transmit it. This problem is even worse for the harder-to-compress text data. The second row of Figures 4 and 5 shows the reverse operation: receiving data via wireless ethernet and decompressing it. The decompression operation is usually less costly

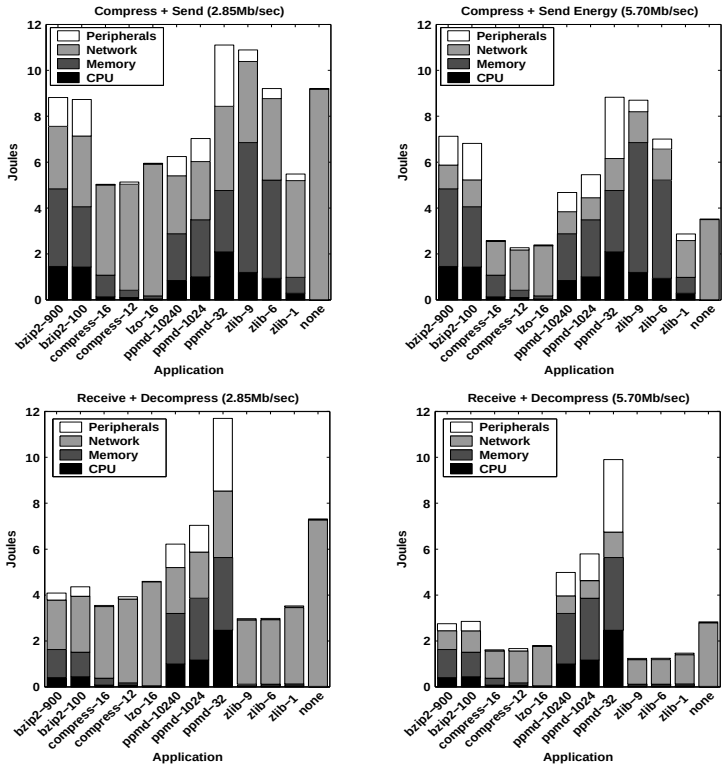

Figure 4. Energy required to transmit 1MB compressible text data

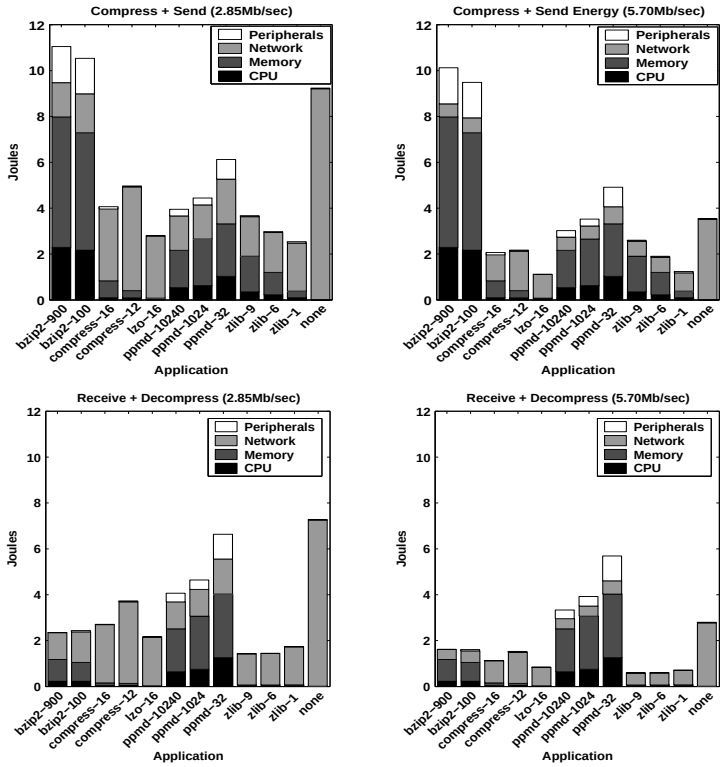

Figure 5. Energy required to transmit 1MB compressible web data

than compression in terms of energy, a fact which will be helpful in choosing a low-energy, asymmetric, lossless compression scheme. As an aside, we have seen that as transmission speed increases, the value of reducing wireless energy through data compression is less. Thus, even when compressing and sending data appears to require the same energy as sending uncompressed data, it is beneficial to apply compression for the greater good: more shared bandwidth will be available to all devices allowing them to send data faster and with less energy. Section 3.3 will discuss how such high net energy is possible despite the motivating observations.

3.3 Energy analysis of popular compressors

We will look deeper into the applications to discover why they cannot exploit the communication - computation energy gap. To perform this analysis, we rely on empirical observations on the Skiff platform as well as the execution-driven simulator known as SimpleScalar [7]. Though SimpleScalar is inherently an out-of-order, superscalar simulator, it has been modified to read statically linked ARM binaries and model the five-stage, in-order pipeline of the SA-110x [2]. As SimpleScalar is beta software we will handle the statistics it reports with caution, using them to explain the *traits* of the compression applications rather than to describe their precise execution on a Skiff. Namely, high instruction counts and high cost of memory access lead to poor energy efficiency.

3.3.1 Instruction count

We begin by looking at the number of instructions each requires to remove and restore a bit (Table 3). The range of instruction counts is one empirical indication of the applications' varying complexity. The excellent performance of *LZO* is due in part to its implementation as a single function, thus there is no function call overhead. In addition, *LZO* avoids superfluous copying due to buffering (in contrast with *compress* and *zlib*). As we will see, the number of memory accesses plays a large role in determining the speed and energy of an application. Each program contains roughly the same percentage of loads and stores, but the great difference in dynamic number of instructions means that programs such as *bzip2* and *PPMd* (each executing over 1 billion instructions) execute more total instructions and therefore have the most memory traffic.

3.3.2 Memory hierarchy

One noticeable similarity of the bars in Figures 4 and 5 is that the memory requires more energy than the processor. To pinpoint the reason for this, microbenchmarks were run on the Skiff memory system.

The SA-110 data cache is 16 KB. It has 32-way associativity and 16 sets. Each block is 32 bytes. Data is evicted at half-block granularity and moves to a 16 entry-by-16 byte write buffer. The write buffer also collects stores that miss in the cache (the cache is writeback/non-write-allocate). The store buffer can merge stores to the same entry.

The hit benchmark accesses the same location in memory in an infinite loop. The miss benchmark consecutively accesses the entire cache with a 32 byte stride followed by the same access pattern offset by 16 KB. Writebacks are measured with a similar pattern, but each load is followed by a store to the same location that dirties the block forcing a writeback the next time that location is read. Store hit energy is subtracted from the writeback energy. The output of the compiler is examined to insure the correct number of load or store instructions is generated. Address generation instructions are ignored for miss benchmarks as their energy is minimal compared to that of a memory access. When measuring store misses in this fashion (with a 32 byte stride), the worse-case behavior of the SA-110's store buffer is exposed as no writes can be combined. In the best case, misses to the the same buffered region can have energy similar to a store hit, but in practice, the majority of store misses for the compression applications are unable to take advantage of batching writes in the store buffer.

Table 4 shows that hitting in the cache requires more energy than an ADD (Table 2), and a cache miss requires up to 145 times the energy of an ADD. Store misses are less expensive as the SA-110 has a store buffer to batch accesses to memory. To minimize energy, then, we must seek to minimize cache-misses which require prolonged access to higher voltage components.

3.3.3 Minimizing memory access energy

One way to minimize misses is to reduce the memory requirements of the application. Figure 6 shows the effect of varying memory size on compression/decompression time and compression ratio. Looking back at Figures 4 and 5, we see the energy implications of choosing the right amount of memory. Most importantly, we see that merely choosing the fastest or best-compressing application does not result in lowest overall energy. Table 5 notes the throughput of each application; we see that with the Skiff's processor, several applications have difficulty meeting the line rate of the network which may preclude their use in latency-critical applications.

In the case of *compress* and *bzip2*, a larger memory footprint stores more information about the data and can be used to improve compression ratio. However, storing more information means less of the data fits in the cache leading to more misses, longer runtime and hence more

	bzip2	compress	LZO	PPMd	zlib
Compress: instructions per bit removed (Text Data)	116	10	7	76	74
Decompress: instructions per bit restored (Text Data)	31	6	2	10	5
Compress: instructions per bit removed (Web Data)	284	9	2	60	23
Decompress: instructions per bit restored (Web Data)	20	5	1	79	3

Table 3. Instructions per bit

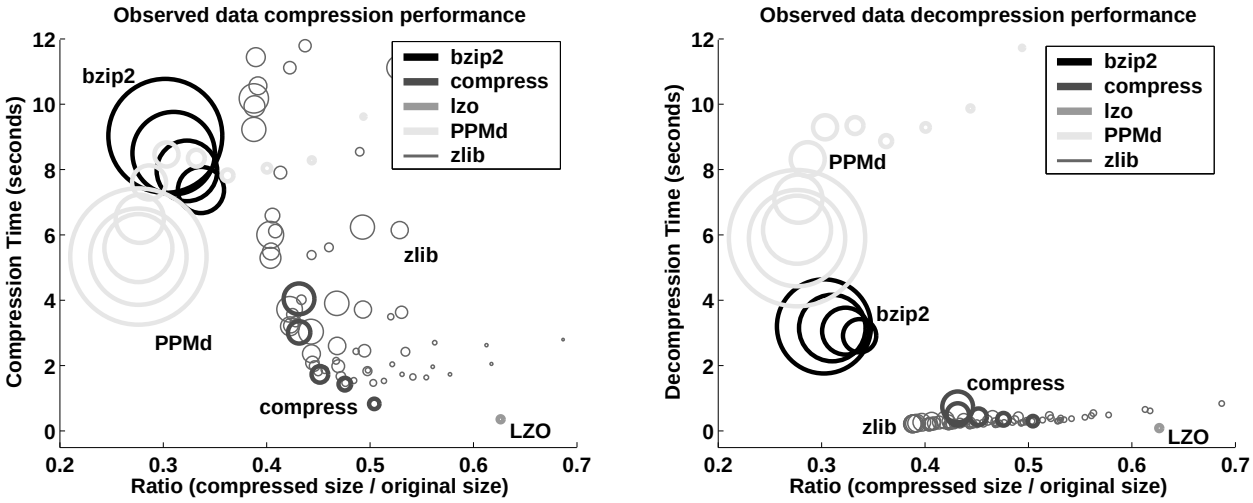

Figure 6. Memory, time, and ratio (Text data). Memory footprint is indicated by area of circle; footprints shown range from 3KB - 8MB

	Cycles	Energy (nJ)
Load Hit	1	2.72
Load Miss	80	124.89
Writeback	107	180.53
Store Hit	1	2.41
Store Miss	33	78.34
ADD	1	0.86

Table 4. Measured memory energy vs. ADD energy

energy. This tradeoff need not apply in the case where more memory allows a more efficient data structure or algorithm. For example, *bzip2* uses a large amount of memory, but for good reason. While we were able to implement its sort with the quicksort routine from the standard C library to save significant memory, the compression takes over 2.5 times as long due to large constants in the runtime of the more traditional quicksort in the standard library. This slowdown occurs even when 16 KB block sizes [38] are used to further reduce memory requirements. Once *PPMd* has enough memory to do useful work, more context information can be stored and less complicated escape handling is necessary.

The widely scattered performance of *zlib*, even with similar footprints, suggest that one must be careful in choosing parameters for this library to achieve the desired goal (speed or compression ratio). Increasing window size effects compression; for a given window, a larger hash table improves speed. Thus, the net effect of more memory is variable. The choice is especially important if memory is constrained as certain window/memory combinations are inefficient for a particular speed or ratio.

The decompression side of the figure underscores the valuable asymmetry of some of the applications. Often decompressing data is a simpler operation than compression which requires less memory (as in *bzip2* and *zlib*). The simple task requires a relatively constant amount of time as there is less work to do: no sorting for *bzip2* and no searching though a history buffer for *zlib*, *LZO*, and *compress* because all the information to decompress a file is explicit. The contrast between compression and decompression for *zlib* is especially large. PPM implementations must go through the same procedure to decompress a file, undoing the arithmetic coding and building a model to keep its probability counts in sync with the compressor's. The arithmetic coder/decoder used in *PPMd* requires more time to decode than encode, so decompression requires more time.

Each of the applications examined allocates fixed-size

	bzip2	compress	LZO	PPMd	zlib
Compress read throughput (Text data)	0.91	3.70	24.22	1.57	0.82
Decompress write throughput (Text data)	2.59	11.65	109.44	1.42	41.15
Compress read throughput (Web data)	0.58	4.15	50.05	2.00	3.29
Decompress write throughput (Web data)	3.25	27.43	150.70	1.75	61.29

Table 5. Application throughputs (Mb/sec)

structures regardless of the input data length. Thus, in several cases more memory is set aside than is actually required. However, a large memory footprint may not be detrimental to an application if its current working set fits in the cache. The simulator was used to gather cache statistics. PPM and BWT are known to be quite memory intensive. Indeed, *PPMd* and *bzip2* access the data cache 1–2 orders of magnitude more often than the other benchmarks. *zlib* accesses data cache almost as much as *PPMd* and *bzip2* during compression, but drops from 150 million accesses to 8.2 million during decompression. Though LZ77 is local by nature, the large window and data structures hurt its cache performance for *zlib* during the compression phase. *LZO* also uses LZ77, but is designed to require just 16KB of memory and goes to main memory over five times less often than the next fastest application. The followup to the SA-110 (the SA-1110 used in Compaq's iPAQ handheld computer) has only an 8KB data cache which would exaggerate any penalties observed here. Though large, low-power caches are becoming possible (the X-Scale has two 32KB caches), as long as the energy of going to main memory remains so much higher, we must be concerned with cache misses.

3.4 Summary

On the Skiff, compression and decompression energy are roughly proportional to execution time. We have seen that the Skiff requires lots of energy to work with aggressively compressed data due to the amount of high-latency/high-power memory references. However using the fastest-running compressor or decompressor is not necessarily the best choice to minimize *total* transmission energy. For example, during decompression both *zlib* and *compress* run slower than *LZO*, but they receive fewer bits due to better compression so total energy is less than *LZO*. These applications successfully walk the tightrope of computation versus communication cost. Despite the greater energy needed to decompress the data, the decrease in receive energy makes the net operation a win. More importantly, we have shown that reducing energy is not as simple as choosing the fastest or best-compressing program.

We can generalize the results obtained on the Skiff in the following fashion. Memory energy is some multiple of CPU energy. Network energy (send and receive) is a far greater multiple of CPU energy. It is difficult to predict how quickly energy of components will change over time. Even predicting whether a certain component's energy usage will grow or shrink can be difficult. Many researchers envision ad-hoc networks made of nearby nodes. Such a topology, in which only short-distance wireless communication is necessary, could reduce the energy of the network interface relative to the CPU and memory. On the other hand, for a given mobile CPU design, planned manufacturing improvements may lower its relative power and energy. Processors once used only in desktop computers are being recast as mobile processors. Though their power may be much larger than that of the Skiff's StrongARM, higher clock speeds may reduce energy. If one subscribes to the belief that CPU energy will steadily decrease while memory and network energy remain constant, then *bzip2* and *PPMd* become viable compressors. If both memory and CPU energy decrease, then current low-energy compression tools (*compress* and *LZO*) can even be surpassed by their computation and memory intensive peers. However, if only network energy decreases while the CPU and memory systems remain static, energy-conscious systems may forego compression altogether as it now requires more energy than transmitting raw data. Thus, it is important for software developers to be aware of such hardware effects if they wish to keep compression energy as low as possible. Awareness of the type of data to be transmitted is important as well. For example, transmitting our world-wide-web data required less energy in general than the text data. Trying to compress pre-compressed data (not shown) requires significantly more energy and is usually futile.

4 Results

We have seen energy can be saved by compressing files before transmitting them over the network, but one must be mindful of the energy required to do so. Compression and decompression energy may be minimized through wise use of memory (including efficient data structures and/or sacrificing compression ratio for cacheability). One must be aware of evolving hardware's effect on overall energy. Finally, knowledge of com-

pression and decompression energy for a given system permits the use of asymmetric compression in which the lowest energy application for compression is paired with the lowest energy application for decompression.

4.1 Understanding cache behavior

Figure 7 shows the compression energy of several successive optimizations of the *compress* program. The baseline implementation is itself an optimization of the original *compress* code. The number preceding the dash refers to the maximum length of codewords. The graph illustrates the need to be aware of the cache behavior of an application in order to minimize energy. The data structure of *compress* consists of two arrays: a hash table to store symbols and prefixes, and a code table to associate codes with hash table indexes. The tables are initially stored back-to-back in memory. When a new symbol is read from the input, a single index is used to retrieve corresponding entries from each array. The "16-merge" version combines the two tables to form an array of structs. Thus, the entry from the code table is brought into the cache when the hash entry is read. The reduction in energy is negligible: though one type of miss has been eliminated, the program is actually dominated by a second type of miss: the probing of the hash table for free entries. The Skiff data cache is small (16KB) compared to the size of the hash table (≈270KB), thus the random indexing into the hash table results in a large number of misses. A more useful energy and performance optimization is to make the hash table more sparse. This admits fewer collisions which results in fewer probes and thus a smaller number of cache misses. As long as the extra memory is available to enable this optimization, about 0.53 Joules are saved compared with applying no compression at all. This is shown by the "16-sparse" bar in the figure. The baseline and "16-merge" implementations require more energy than sending uncompressed data. A 12-bit version of compress is shown as well. Even when peripheral overhead energy is disregarded, it outperforms or ties the 16-bit schemes as its reduced memory energy due to fewer misses makes up for poorer compression.

Another way to reduce cache misses is to fit both tables completely in the cache. Compare the following two structures:

```
struct entry{
  int fcode;
  unsigned short code;
}table[SIZE];
```

```
struct entry{
  signed fcode:20;
  unsigned code:12;
}table[SIZE];
```

Each `entry` stores the same information, but the array on the left wastes four bytes per entry. Two bytes are used only to align the short `code`, and overly-wide

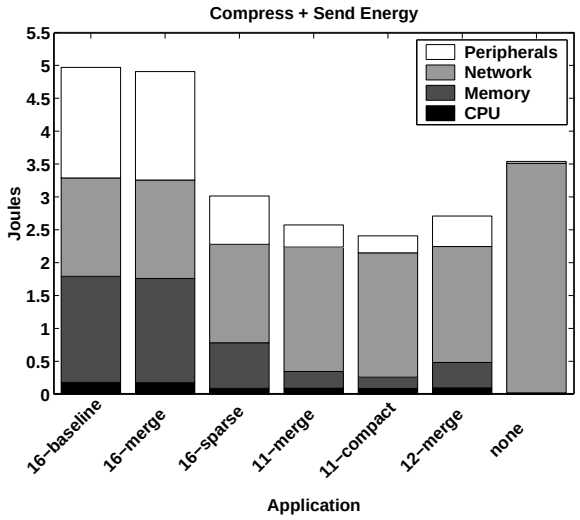

Figure 7. Optimizing *compress* (Text data)

types result in twelve wasted bits in `fcode` and four bits wasted in `code`. Using bitfields, the layout on the right contains the same information yet fits in half the space. If the entry were not four bytes, it would need to contain more members for alignment. Code with such structures would become more complex as C does not support arrays of bitfields, but unless the additional code introduces significant instruction cache misses, the change is low-impact. A bitwise AND and a shift are all that is needed to determine the offset into the compact structure. By allowing the whole table to fit in the cache, the program with the compacted array has just 56,985 data cache misses compared with 734,195 in the un-packed structure; a 0.0026% miss rate versus 0.0288%. The energy benefit for *compress* with the compact layout is negligible because there is so little CPU and memory energy to eliminate by this technique. The "11-merge" and "11-compact" bars illustrate the similarity. Nevertheless, 11-compact runs 1.5 times faster due to the reduction in cache misses, and such a strategy could be applied to any program which needs to reduce cache misses for performance and/or energy. Eleven bit codes are necessary even with the compact layout in order to reduce the size of the data structure. Despite a dictionary with half the size, the number of bytes to transmit increases by just 18% compared to "12-merge." Energy, however, is lower with the smaller dictionary due to less energy spent in memory and increased speeds which reduce peripheral overhead.

4.2 Exploiting the sleep mode

It has been noted that when a platform has a low-power idle state, it may be sensible to sacrifice energy

in the short-term in order to complete an application quickly and enter the low-power idle state [26]. Figure 8 shows the effect of this analysis for compression and sending of text. Receive/decompression exhibits similar, but less-pronounced variation for different idle powers. It is interesting to note that, assuming a low-power idle mode can be entered once compression is complete, one's choice of compression strategies will vary. With its 1 Watt of idle power, the Skiff would benefit most from *zlib* compression. A device which used negligible power when idle would choose the *LZO* compressor. While *LZO* does not compress data the most, it allows the system to drop into low-power mode as quickly as possible, using less energy when long idle times exist. For web data (not shown due to space constraints) the compression choice is *LZO* when idle power is low. When idle power is one Watt, *bzip2* energy is over 25% more energy efficient than the next best compressor.

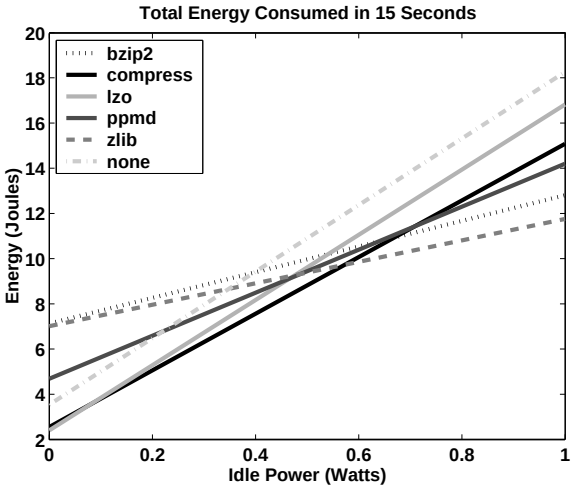

Figure 8. Compression + Send energy consumption with varying sleep power (Text data)

4.3 Asymmetric compression

Consider a wireless client similar to the Skiff exchanging English text with a server. All requests by the client should be made with its minimal-energy compressor, and all responses by the server should be compressed in such a way that they require minimal decompression energy at the client. Recalling Figures 4 and 5, and recognizing that the Skiff has no low-power sleep mode, we choose "compress-12" (the twelve-bit codeword LZW compressor) for our text compressor as it provides the lowest total compression energy over all communication speeds.

To reduce decompression energy, the client can request data from the server in a format which facilitates low-energy decompression. If latency is not critical and the client has a low-power sleep mode, it can even wait while the server converts data from one compressed format to another. On the Skiff, *zlib* is the lowest energy decompressor for both text and web data. It exhibits the property that regardless of the effort and memory parameters used to compress data, the resulting file is quite easy to decompress. The decompression energy difference between *compress*, *LZO*, and *zlib* is minor at 5.70 Mb/sec, but more noticeable at slower speeds.

Figure 9 shows several other combinations of compressor and decompressor at 5.70 Mb/sec. "zlib-9 + zlib-9" represents the symmetric pair with the least decompression energy, but its high compression energy makes it unlikely to be used as a compressor for devices which must limit energy usage. "compress-12 + compress-12" represents the symmetric pair with the least compression energy. If symmetric compression and decompression is desired, then this "old-fashioned" Unix compress program can be quite valuable. Choosing "zlib-1" at both ends makes sense as well – especially for programs linked with the zlib library. Compared with the minimum symmetric compressor-decompressor, asymmetric compression on the Skiff saves only 11% of energy. However, modern applications such as ssh and mod_gzip use "zlib-6" at both ends of the connection. Compared to this common scheme, the optimal asymmetric pair yields a 57% energy savings – mostly while performing compression.

It is more difficult to realize a savings over symmetric zlib-6 for web data as all compressors do a good job compressing it and "zlib-6" is already quite fast. Nevertheless, by pairing "lzo" and "zlib-9," we save 12% of energy over symmetric "lzo" and 31% over symmetric "zlib-6."

5 Related work

This section discusses data compression for low-bandwidth devices and optimizing algorithms for low energy. Though much work has gone into these fields individually, it is difficult to find any which combines them to examine lossless data compression from an energy standpoint. Computation-to-communication energy ratio has been been examined before [12], but this work adds physical energy measurements and applies the results to lossless data compression.

5.1 Lossless Data compression for low-bandwidth devices

Like any optimization, compression can be applied at many points in the hardware-software spectrum. When

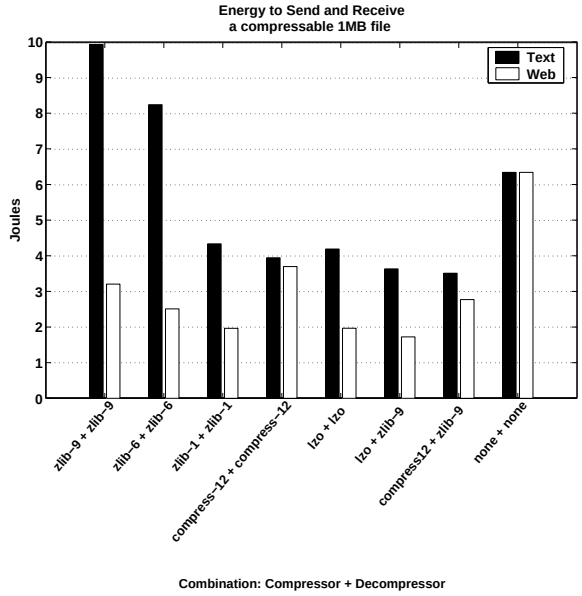

Figure 9. Choosing an optimal compressor-decompressor pair

applied in hardware, the benefits and costs propagate to all aspects of the system. Compression in software may have a more dramatic effect, but for better or worse, its effects will be less global.

The introduction of low-power, portable, low-bandwidth devices has brought about new (or rediscovered) uses for data compression. Van Jacobson introduced TCP/IP Header Compression in RFC1144 to improve interactive performance over low-speed (wired) serial links [19], but it is equally applicable to wireless. By taking advantage of uniform header structure and self-similarity over the course of a particular networked conversation, 40 byte headers can be compressed to 3–5 bytes. Three byte headers are the common case. An all-purpose header compression scheme (not confined to TCP/IP or any particular protocol) appears in [24]. TCP/IP payloads can be compressed as well with IP-Comp [39], but this can be wasted effort if data has already been compressed at the application layer.

The Low-Bandwidth File System (LBFS) exploits similarities between the data stored on a client and server, to exchange only data blocks which differ [31]. Files are divided into blocks with content-based fingerprint hashes. Blocks can match any file in the file system or the client cache; if client and server have matching block hashes, the data itself need not be transmitted. Compression is applied before the data is transmitted. Rsync [44] is a protocol for efficient file transfer which preceded LBFS. Rather than content-based fingerprints, Rsync uses its rolling hash function to account for changes in block size. Block hashes are compared for a pair of files to quickly identify similarities between client and server. Rsync block sharing is limited to files of the same name.

A protocol-independent scheme for text compression, NCTCSys, is presented in [30]. NCTCSys involves a common dictionary shared between client and server. The scheme chooses the best compression method it has available (or none at all) for a dataset based on parameters such as file size, line speed, and available bandwidth.

Along with remote proxy servers which may cache or reformat data for mobile clients, splitting the proxy between client and server has been proposed to implement certain types of network traffic reduction for HTTP transactions [14, 23]. Because the delay required for manipulating data can be small in comparison with the latency of the wireless link, bandwidth can be saved with little effect on user experience. Alternatively, compression can be built into servers and clients as in the *mod_gzip* module available for the Apache webserver and HTTP 1.1 compliant browsers [16]. Delta encoding, the transmission of only parts of documents which differ between client and server, can also be used to compress network traffic [15, 27, 28, 35].

5.2 Optimizing algorithms for low energy

Advanced RISC Machines (ARM) provides an application note which explains how to write C code in a manner best-suited for its processors and ISA [1]. Suggestions include rewriting code to avoid software emulation and working with 32 bit quantities whenever possible to avoid a sign-extension penalty incurred when manipulating shorter quantities. To reduce energy consumption and improve performance, the OptAlg tool represents polynomials in a manner most efficient for a given architecture [34]. As an example, cosine may be expressed using two MAC instructions and an MUL to apply a “Horner transform” on a Taylor Series rather than making three calls to a cosine library function.

Besides architectural constraints, high level languages such as C may introduce false dependencies which can be removed by disciplined programmers. For instance, the use of a global variable implies loads and stores which can often be eliminated through the use of register-allocated local variables. Both types of optimizations are used as guidelines by PHiPAC [6], an automated generator of optimized libraries. In addition to these general coding rules, architectural parameters are provided to a code generator by search scripts which work to find the best-performing routine for a given platform.

Yang et al. measured the power and energy impact of various compiler optimizations, and reached the conclusion that energy can be saved if the compiler can reduce

execution time and memory references [48]. Šimunić found that floating point emulation requires much energy due to the sheer number of extra instructions required [46]. It was also discovered that instruction flow optimizations (such as loop merging, unrolling, and software pipelining) and ISA specific optimizations (e.g., the use of a multiply-accumulate instruction) were not applied by the ARM compiler and had to be introduced manually. Writing such energy-efficient source code saves more energy than traditional compiler speed optimizations [45].

The CMU Odyssey project studied "application-aware adaptation" to deal with the varying, often limited resources available to mobile clients. Odyssey trades data quality for resource consumption as directed by the operating system. By placing the operating system in charge, Odyssey balances the needs of all running applications and makes the choice best suited for the system. Application-specific adaptation continues to improve. When working with a variation of the Discrete Cosine Transform and computing first with DC and low-frequency components, an image may be rendered at 90% quality using just 25% of its energy budget [41]. Similar results are shown for FIR filters and beamforming using a most-significant-first transform. Parameters used by JPEG lossy image compression can be varied to reduce bandwidth requirements and energy consumption for particular image quality requirements [43]. Research to date has focused on situations where energy-fidelity tradeoffs are available. Lossless compression does not present this luxury because the original bits must be communicated in their entirety and re-assembled in order at the receiver.

6 Conclusion and Future Work

The value of this research is not merely to show that one can optimize a given algorithm to achieve a certain reduction in energy, but to show that the choice of how and whether to compress is not obvious. It is dependent on hardware factors such as relative energy of CPU, memory, and network, as well as software factors including compression ratio and memory access patterns. These factors can change, so techniques for lossless compression prior to transmission/reception of data must be re-evaluated with each new generation of hardware and software. On our StrongARM computing platform, measuring these factors allows an energy savings of up to 57% compared with a popular default compressor and decompressor. Compression and decompression often have different energy requirements. When one's usage supports the use of asymmetric compression and decompression, up to 12% of energy can be saved compared with a system using a single optimized application for both compression and decompression.

When looking at an entire system of wireless devices, it may be reasonable to allow some to individually use more energy in order to minimize the total energy used by the collection. Designing a low-overhead method for devices to cooperate in this manner would be a worthwhile endeavor. To facilitate such dynamic energy adjustment, we are working on EProf: a portable, realtime, energy profiler which plugs into the PC-Card socket of a portable device [22]. EProf could be used to create feedback-driven compression software which dynamically tunes its parameters or choice of algorithms based on the measured energy of a system.

7 Acknowledgements

Thanks to John Ankcorn, Christopher Batten, Jamey Hicks, Ronny Krashinsky, and the anonymous reviewers for their comments and assistance. This work is supported by MIT Project Oxygen, DARPA PAC/C award F30602-00-2-0562, NSF CAREER award CCR-0093354, and an equipment grant from Intel.

References

[1] Advanced RISC Machines Ltd (ARM). *Writing Efficient C for ARM*, Jan. 1998. Application Note 34.

[2] T. M. Austin and D. C. Burger. SimpleScalar version 4.0 release. *Tutorial in conjunction with 34th Annual International Symposium on Microarchitecture*, Dec. 2001.

[3] T. Bell and D. Kulp. Longest match string searching for Ziv-Lempel compression. Technical Report 06/89, Department of Computer Science, University of Canterbury, New Zealand, 1989.

[4] T. Bell, M. Powell, J. Horlor, and R. Arnold. The Canterbury Corpus. http://www.corpus.canterbury.ac.nz/.

[5] T. Bell, I. H. Witten, and J. G. Cleary. Modeling for text compression. *ACM Computing Surveys*, 21(4):557–591, 1989.

[6] J. Bilmes, K. Asanović, C.-W. Chin, and J. Demmel. Optimizing matrix multiply using PHiPAC: a portable, high-performance, ANSI C coding methodology. In *11th ACM International Conference on Supercomputing*, July 1997.

[7] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[8] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital Systems Research Center, May 1994.

[9] J. Gailly and M. Adler. zlib. http://www.gzip.org/zlib.

[10] J. Gailly, Maintainer. `comp.compression` Internet newsgroup: Frequently Asked Questions, Sept. 1999.

[11] J. Gilchrist. Archive comparison test. http://compression.ca.

[12] P. J. Havinga. Energy efficiency of error correction on wireless systems. In *IEEE Wireless Communications and Networking Conference*, Sept. 1999.

[13] J. Hicks et al. Compaq personal server project, 1999. http://crl.research.compaq.com /projects/personalserver/default.htm.

[14] B. C. Housel and D. B. Lindquist. Webexpress: a system for optimizing web browsing in a wireless environment. In *Proceedings of the Second Annual International Conference on Mobile Computing and Networking*, 1996.

[15] J. J. Hunt, K.-P. Vo, and W. F. Tichy. An empirical study of delta algorithms. In *Software configuration management: ICSE 96 SCM-6 Workshop*. Springer, 1996.

[16] Hyperspace Communications, Inc. Mod_gzip. http://www.ehyperspace.com /htmlonly/products/mod_gzip.html.

[17] Intel Corporation. *SA-110 Microprocessor Technical Reference Manual*, December 2000.

[18] Intel Corporation. *Intel StrongARM SA-1110 Microprocessor Developer's Manual*, October 2001.

[19] V. Jacobson. RFC 1144: Compressing TCP/IP headers for low-speed serial links, Feb. 1990.

[20] K. Jamieson. Implementation of a power-saving protocol for ad hoc wireless networks. Master's thesis, Massachusetts Institute of Technology, Feb. 2002.

[21] P. Jannesen et. al. (n)compress. available, among other places, in Redhat 7.2 distribution of Linux.

[22] K. Koskelin, K. Barr, and K. Asanović. Eprof: An energy profiler for the iPaq. In *2nd Annual Student Oxygen Workshop*. MIT Project Oxygen, 2002.

[23] R. Krashinsky. Efficient web browsing for mobile clients using HTTP compression. Technical Report MIT-LCS-TR-882, MIT Lab for Computer Science, Jan. 2003.

[24] J. Lilley, J. Yang, H. Balakrishnan, and S. Seshan. A unified header compression framework for low-bandwidth links. In *6th ACM MOBICOM*, Aug. 2000.

[25] Lycos. Lycos 50, Sept. 2002. Top 50 searches on Lycos for the week ending September 21, 2002.

[26] A. Miyoshi, C. Lefurgy, E. V. Hensbergen, R. Rajamony, and R. Rajkumar. Critical power slope: Understanding the runtime effects of frequency scaling. In *International Conference on Supercomputing*, June 2002.

[27] J. C. Mogul. Trace-based analysis of duplicate suppression in HTTP. Technical Report 99.2, Compaq Computer Corporation, Nov. 1999.

[28] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. Technical Report 97/4a, Compaq Computer Corporation, Dec. 1997.

[29] J. Montanaro et al. A 160-mhz, 32-b, 0.5-w CMOS RISC microprocessor. *IEEE Journal of Solid-State Circuits*, 31(11), Nov. 1996.

[30] N. Motgi and A. Mukherjee. Network conscious text compression systems (NCTCSys). In *Proceedings of International Conference on Information and Theory: Coding and Computing*, 2001.

[31] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 174–187, Chateau Lake Louise, Banff, Canada, October 2001.

[32] Nielsen NetRatings Audience Measurement Service. Top 25 U.S Properties; Week of Sept 15th., Sept. 2002.

[33] M. F. Oberhumer. LZO. http://www.oberhumer.com/opensource/lzo/.

[34] A. Peymandoust, T. Šimunić, and G. D. Micheli. Low power embedded software optimization using symbolic algebra. In *Design, Automation and Test in Europe*, 2002.

[35] J. Santos and D. Wetherall. Increasing effective link bandwidth by suppressing replicated data. In *USENIX Annual Technical Conference*, June 1998.

[36] K. Sayood. *Introduction to data compression*. Morgan Kaufman Publishers, second edition, 2002.

[37] J. Seward. bzip2. http://www.spec.org /osg/cpu2000/CINT2000/256.bzip2/docs/256.bzip2.html.

[38] J. Seward. e2comp bzip2 library. http://cvs.bofh.asn.au/e2compr/index.html.

[39] A. Shacham, B. Monsour, R. Pereira, and M. Thomas. RFC 3173: IP payload compression protocol, Sept. 2001.

[40] D. Shkarin. PPMd. ftp://ftp.elf.stuba.sk/pub/pc/pack/ppmdi1.rar.

[41] A. Sinha, A. Wang, and A. Chandrakasan. Algorithmic transforms for efficient energy scalable computation. In *IEEE International Symposium on Low Power Electronics and Design*, August 2000.

[42] Standard Performance Evaluation Corporation. CPU2000, 2000.

[43] C. N. Taylor and S. Dey. Adaptive image compression for wireless multimedia communication. In *IEEE International Conference on Communication*, June 2001.

[44] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, Apr. 2000.

[45] T. Šimunić, L. Benini, and G. D. Micheli. Energy-efficient design of battery-powered embedded systems. In *IEEE International Symposium on Low Power Electronics and Design*, 1999.

[46] T. Šimunić, L. Benini, G. D. Micheli, and M. Hans. Source code optimization and profiling of energy consumption in embedded systems. In *International Symposium on System Synthesis*, 2000.

[47] M. A. Viredaz and D. A. Wallach. Power evaluation of Itsy version 2.4. Technical Report TN-59, Compaq Computer Corporation, February 2001.

[48] H. Yang, G. R. Gao, A. Marquez, G. Cai, and Z. Hu. Power and energy impact of loop transformations. In *Workshop on Compilers and Operating Systems for Low Power 2001, Parallel Architecture and Compilation Techniques*, Sept. 2001.

APPENDIX C
Fine-Grain CAM-Tag Cache Resizing Using Miss Tags

Michael Zhang
MIT Laboratory for Computer Science
200 Technology Square
Cambridge, MA 02139
rzhang@cag.lcs.mit.edu

Krste Asanović
MIT Laboratory for Computer Science
200 Technology Square
Cambridge, MA 02139
krste@cag.lcs.mit.edu

ABSTRACT

A new dynamic cache resizing scheme for low-power CAM-tag caches is introduced. A control algorithm that is only activated on cache misses uses a duplicate set of tags, the *miss tags*, to minimize active cache size while sustaining close to the same hit rate as a full size cache. The cache partitioning mechanism saves both switching and leakage energy in unused partitions with little impact on cycle time. Simulation results show that the scheme saves 28–56% of data cache energy and 34–49% of instruction cache energy with minimal performance impact.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*Associative Memory, Cache Memory, Primary Memory*

General Terms

Design

Keywords

Content-Addressable-Memory, Low-Power, Cache Resizing, Energy Efficiency, Leakage Current

1. INTRODUCTION

Energy dissipation has emerged as one of the primary constraints for microprocessor designers. In most microprocessor designs, caches dissipate a significant fraction of total power. For example, the Alpha 21264 dissipates 16% [12] and the StrongArm dissipates more than 43% [19] of overall power in caches. As a result, there has been great interest in reducing cache power consumption.

Initial cache energy reduction techniques focused on dynamic switching power [1, 2, 3, 4, 7, 10, 13, 22]. With technology scaling, leakage current is increasing exponentially, and more attention has been paid to leakage power reduction [9, 11, 15, 16, 18, 20].

ISLPED'02, August 12-14, 2002, Monterey, California, USA.

One approach for reducing cache power consumption is cache resizing, where the active size of the cache is reduced to match the current working set. Previously reported cache resizing schemes can be categorized by the mechanism used to activate and deactivate cache entries, and by the control policy used to select the active partition. Some schemes deactivate cache entries line by line [9, 11], while others deactivate the cache by sets, ways, or both [1, 16, 20]. The control policy used to select the active set can be off-line, where the working set is statically determined by profiling the application [1], or on-line, where the working set is dynamically determined as the application executes [9, 11, 16, 20].

Previous cache resizing techniques are designed for RAM-tag caches, where cache tags are held in RAM structures. However, commercial low-power microprocessors use CAM-tag caches, where the cache tags are held in Content Addressable Memory [14, 19]. CAM-tag caches are popular in low-power processors because they provide high associativity, which avoids expensive cache misses, and results in lower overall energy [23].

This paper introduces *miss tag resizing* (MTR), a new cache resizing scheme for CAM-tag caches. MTR uses hierarchical bitlines to divide each cache subbank into small way partitions, such that switching and leakage power is only dissipated in active ways. In addition, individual cache lines within an active partition can be disabled to further reduce leakage power. Because CAM-tag caches have high associativity (32-way for the design simulated), partitioning the cache by way gives much finer grain control over cache size compared to RAM-tag way activation [1]. It also avoids the data remapping problem inherent in set resizing schemes [16]. In addition, the scheme proposed here adapts associativity independently in each sub-bank, thereby allowing total cache size to be varied a single line at a time. Resizing of different subbanks is spaced evenly in time so that at most a single dirty line needs to be written back for a resize event.

The size of an MTR cache is governed using an on-line control policy which aims to reduce the cache size to the smallest value that will give a minimal miss rate increase compared to the full sized cache. The control policy uses an extra set of tags, the *miss tags*, which are only accessed on misses to determine if a full-sized cache would have hit. Because the miss tags are only accessed on misses, they add no additional switching energy to hits and can be implemented using slower, denser, and less leaky transistors, e.g., high V_T or long channel transistors. The main penalty for using miss

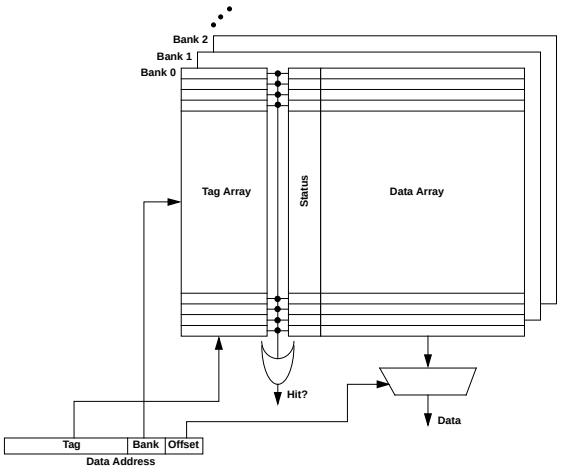

Figure 1: CAM-tag cache organization.

tags is the additional area overhead, which we estimate at around 10% depending on actual layout styles.

The rest of the paper is organized as follows. Section 2 reviews related work on cache resizing. Section 3 presents the MTR algorithm. Section 4 describes the hardware modifications for energy reduction. Section 5 gives results for active cache size reductions. Section 6 presents the energy savings achieved by MTR. And Section 7 concludes.

2. RELATED WORK

In this section, we discuss existing cache resizing techniques and cache line deactivation techniques. An off-line resizing technique was proposed in [1]. Applications are profiled prior to execution to determine an optimal set-associativity. At run-time, cache ways of the L1 RAM-tag set-associative cache are turned off according to the profile information. This technique reduces both switching and leakage energy by powering down the entire cache way. However, it does not adapt to varying cache usage during different phases of the program execution. As we will show later, many benchmarks have working sets that vary widely during various phases of execution. Furthermore, these static techniques do not work well for multi-programmed machines, where working set size also varies as a function of the active process. The DRI I-cache [16] is an on-line resizing technique that resizes a RAM-tag instruction cache by measuring the miss rate and keeping it under a preset threshold. This performance threshold is set to a typical cache miss rate prior to execution, which does not adapt to program execution phases. Line deactivation techniques are similar to the above resizing techniques. These techniques usually turn off individual cache lines that are not necessarily contiguous. In cache decay [11], a per-line counter tracks the usage of each cache line. Lines with no recent uses are turned off. This technique eliminates the static energy of dead lines but does not reduce switching energy. Adaptive mode control (AMC) [9] resizes a RAM-tag cache using a technique similar to cache decay. AMC keeps all tags turned on. An ideal miss rate is obtained by searching the entire tag array, and an actual miss rate is obtained by only searching the tags of all the active lines. When these two miss rates differ by more than a preset performance factor, the resize

```
cache_access(action, addr_tag, addr_offset, data) {
  if (addr_tag in tag_array) {   /* hit case */
    if (action == Read) {
      return data_array[addr_tag, addr_offset];
    } else {
      data_array[addr_tag, addr_offset] = data;
    }
    return hit;
  } else { /* miss case */
    /* fetch data from L2 and update the cache */
    fetch_from_memory(addr_tag, addr_offset);
    /* check whether tag is in MTR tag array */
    if (addr_tag in MTR_tag_array) {
      /* if tag is found in MTR, */
      /* increment MTR hit counter */
      MTR_hits++;
    } else {
      /* otherwise, write the tag into MTR array */
      update_MTR_tag_content(addr_tag);
    }
    return miss;
  }
}

cache_resize() {
  if (MTR_hits > HI_BOUND) {
    upsize();
  } else if (MTR_hits < LO_BOUND) {
    downsize();
  } else {
    do_nothing();
  }
  /* reset the MTR hit counter for */
  /* next resizing interval */
  MTR_hits = 0;
}
```

Figure 2: Pseudo-code for MTR.

interval is adjusted. This technique eliminates the need for presetting the desired miss rates, but only reduces leakage power in the data arrays. Tag array lookup, however, is a significant portion of the cache access energy, especially for CAM-tag caches. In [20], various design choices are compared to evaluate the usefulness of resizable caches. On average, over 50% cache size reduction is achieved with either selective ways [1] or selective sets [16]. Turning off portions of the cache generally discards the stored data, thus increasing miss rate and the number of L2 accesses. In [8], the effect of L2 energy overhead is examined. Our MTR scheme is similar to AMC in that we resize based on the difference between the full cache hit rate and the reduced cache hit rate. However, we employ a separate set of tags that are only accessed on misses to gather the full cache hit rate. This avoids additional switching and leakage power in the regular CAM tags. Also, we use the miss rate difference to control a fine-grain partitionable cache which can save switching as well as leakage power. Another problem with previous partitioning schemes is that when applied to a data cache, they can generate a large number of dirty line writebacks in a short time interval when a set or way is deactivated, or when a decay interval elapses. These write back bursts add to cache control complexity and can cause additional performance degradation. MTR performs way deactivation within a highly associative cache one line at a time, thus avoids write back bursts.

3. MISS-TAG RESIZING TECHNIQUE

Figure 1 shows a typical CAM-tag cache organization. The entire cache is divided into subbanks, each consisting of a tag array and a data array, where a subbank is a cache *set*. Within each set are the cache *ways*. The tags are stored in CAM structures to give high associativity at low power. During each cache access, one subbank (set) of the cache is accessed and the tag is broadcast to the entire tag array. A matched tag results in a hit and triggers the appropriate wordline to enable the access.

To implement MTR, we add an extra set of tags, the *miss-tags*, which act as the tags of a fixed-size cache. These tags keep track of what the cache contents would have been if the cache was always full size. During a regular cache miss, we consult the miss-tag arrays to see whether having a full cache could have avoided the miss. A per-subbank counter is used to record the number of miss-tag hits, which is precisely the difference between the number of misses in the down-sized cache and in a full size cache. A large difference in the miss rates suggests that having a larger cache will reduce the miss rate; a small difference indicates that perhaps a smaller cache would be adequate. Two scenarios could explain a small difference in miss rate between the full size and reduced size caches. First, there are no misses in the regular tags, indicating that the program has a small working set. In the second scenario, there are many misses in the regular tags, most of which also miss in the miss tags. This suggests that the program has little temporal locality, such as a data streaming application.

The resizing decision is based on the difference in miss rates between the active tags and the miss tags. The pseudo-code in Figure 2 illustrates the resizing control loop of MTR. There are three parameters in the MTR scheme: *miss lower bound*, *miss upper bound*, and *resize interval*. In Section 5.2, we will discuss the choices of resizing parameters in detail. Each subbank is independently resized once during each resizing interval. Resizing events are spread out evenly within each interval so that only one subbank resizes at a time to minimize writeback traffic burst to the lower levels of the memory hierarchy.

4. HARDWARE MODIFICATION

Figure 3 details three circuit techniques used by MTR. For the SRAM cells in both data and tag arrays, we use the Gated-Vdd technique [15] to reduce leakage energy by adding an N-type stack transistor. When signal `Line_On` is turned off, it virtually eliminates leakage current in the SRAM cells. We also use the leakage-biased bitline (LBB) technique proposed in [17] to reduce the leakage in SRAM bitlines, CAM bitlines and search lines, and CAM match lines. The leakage power of the circuit depends on the actual voltage of these heavily capacitive lines. The LBB technique turns off the precharge of these lines, allowing them to self-bias their voltage levels to the optimal values, at which leakage power is minimized using leakage currents. The cache subbanks are divided into eight equal partitions using hierarchical bitlines [7]. The `Partition_On` bits are used to control the activation of each partition. An inactive partition consumes no switching energy and minimal leakage energy.

Since the miss-tags are only used during a cache miss, we can use slow, low-leakage components without incurring

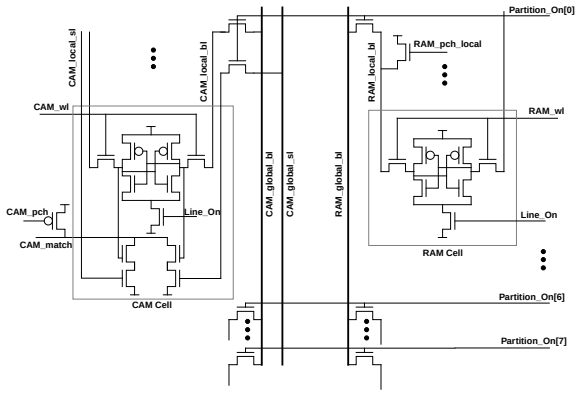

Figure 3: Energy reduction techniques used by MTR: Gated-Vdd for SRAM cell leakage reduction; Leakage-Bias for CAM match line; hierarchical bitlines for subbank partitioning.

delay overhead. The energy overhead of miss-tag accesses is added to L2 access energy and is discussed in Section 6. The area overhead can be reduced by using a denser layout for the tags, for example, adopting a hybrid RAM-CAM structure to reduce the number of match comparators.

5. CACHE SIZE REDUCTION RESULTS

In order to evaluate MTR, we modified the SimpleScalar [5] simulator. We modeled an in-order single issue core in our experiments. The benchmark set we used is a subset of `SpecINT2000` and `SpecFP2000`, each running for 1.5 billion cycles with the reference inputs. We chose a typical low-power cache configuration [14] as a baseline. It is a 32KB cache implemented in 32 1KB subbanks. Each subbank consists of 32 cache lines of 32 bytes. The cache is 32-way set-associative with a FIFO replacement policy in each subbank.

One unary encoded resizing pointer per subbank is used to control which cache lines to activate/deactivate, similar to the XScale FIFO pointer [14]. When a cache is downsized, only the last active line is turned off. When it is upsized, however, the entire partition where the last active line resides is turned on. If all the lines in the entire partition are already active, the next partition is turned on. When all the lines in a partition are inactive, the partition is turned off. To avoid thrashing with small cache sizes, we set the minimum cache size to be one partition.

5.1 Baseline Case

We implemented a baseline resizing technique to compare against the miss tags scheme. This baseline technique works exactly like MTR *except* it compares the *actual cache miss rate* with the miss bounds to make resizing decisions, similar to DRI I-cache [16]. We will refer to this baseline technique as *Miss-Rate-Based-Resizing* (MRBR).

5.2 Impact of Resizing Parameters

From simulation results, we found that no individual parameter has a large impact on resizing performance. The most important parameter, rather, is the ratio of the miss upper/lower bounds to the resize interval. For example, setting the miss bound of 5 to 10 misses for a 32k resizing interval yields similar results for a range of 10 to 20 misses for

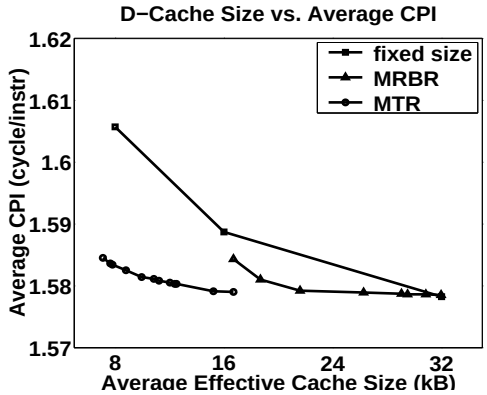

Figure 4: CPI versus effective cache size for L1 data cache. MTR gives the smallest effective cache size for a given CPI.

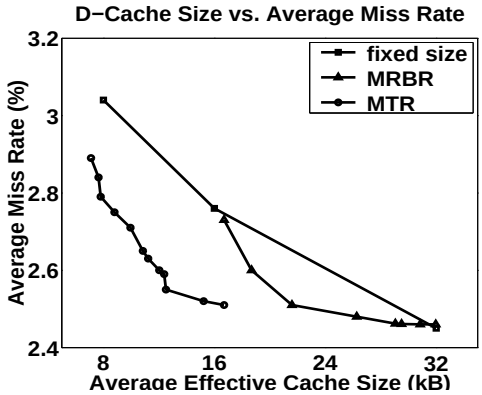

Figure 5: Miss Rate versus effective cache size for L1 data cache. MTR gives the smallest effective cache size for a given miss rate.

a 64k resizing interval. Simulations show that for larger resize intervals, the number of writebacks decrease. However, when the resize interval is too large, MTR starts to yield sub-optimal results. We have found that resize intervals of 128K references worked well for the benchmarks studied, i.e., resize every 128k memory references.

5.3 Data Cache Resizing Results

Figure 4 shows the resizing results for the L1 data cache. Each data point (effective cache size and CPI pair) is obtained by varying the miss bounds and resizing interval length to obtain the optimal CPI for a given effective cache size. Average cache size is calculated by averaging the percentage of active partitions in each resizing period. In order to verify that both resizing techniques work better than a fixed-size cache, we simulated the CPI of fixed-size caches of sizes 32KB, 16KB, and 8KB. This figure shows that for the same CPI, MTR yields much smaller effective cache sizes. We limited ourselves to considering configurations that yield less than a 2% CPI increase to ensure MTR does not incur a large performance penalty. Parameters were varied to show the trade off between effective cache size and performance. For the same effective cache size, MTR performs much better than the baseline technique. Figure 5 further supports the above result. MTR introduces less than a 16% increase in the largest fixed cache miss rate. Again, for the same effective cache size, MTR has the lowest miss rate. On average, MTR uses less than an 8KB effective cache size while increasing the CPI by less than 1.5%.

Figure 6 shows how the effective cache size and the actual miss rates change over time with MTR. The figures on the left-hand side show the effective cache size over time. We observe two different behaviors. Benchmarks `164.gzip`, `177.mesa`, `183.equake`, `197.parser`, and `256.bzip2` demonstrate MTR's ability to adapt to different phases of the execution with varying cache usage. For the rest of the benchmarks, cache usage is constant throughout the execution. MTR is able to find the optimal size for each benchmark without prior profiling information. The figures on the right-hand-side show how the miss rates change throughout the execution. We observe that an increase in the miss rate is countered by an increase in cache size, which in return, reduces miss rate.

5.4 Instruction Cache Resizing Results

For our benchmark set, the instruction cache has extremely low miss rates. Therefore, it is easier to find a common reference miss rate for a large set of benchmarks. For all the benchmarks we used in this paper, the baseline resizing technique and MTR have similar performance. Both of them outperform the fixed size instruction cache. Figures 9 and 8 show that MTR uses an effective cache size of less than 12KB while introducing, on average, less than 12% increase in miss rate and 1.4% increase in CPI.

6. ENERGY REDUCTION RESULTS

In this section, we present the energy savings obtained by MTR. The energy consumption figures are obtained through HSpice simulation of extracted layout from Cadence [6] using TSMC 0.25μm technology [21]. The cache design has been significantly optimized for low power, including divided word lines and low-swing bitlines. Table 1 shows the different energy components of this CAM-tag cache. MTR reduces the data array and CAM-tag array access energy but not decoding energy. Since the actual percentage of cache leakage power in the total cache power can vary significantly due to process technology, operating temperatures and voltages, among other factors, we quantify cache leakage as a percentage of total cache power, and demonstrate the savings across a range of possible values. We perform a similar sensitivity analysis for L2 cache energy by quantifying L2 access energy as a multiple of L1 access energy and give results for a range of values. We include the search energy for the miss-tags as part of L2 energy. The energy reduction is calculated as

$$
\begin{array}{ll}
 & \text{L1 switching energy reduction} \times \% \text{ of switching energy} \\
+ & \text{L1 leakage energy reduction} \times \% \text{ of leakage energy} \\
- & \text{Miss Rate Increase} \times \text{L2 access energy}
\end{array}
$$

Figures 10 and 11 show the energy reduction of data and instruction cache. The x-axis represents the percentage of leakage energy in the total energy consumption. The y-axis represents the energy savings. From previous experiments, we use resizing parameters such that the effective data cache size is 8KB and effective instruction cache is 12KB. These parameters are chosen to minimize the performance impact

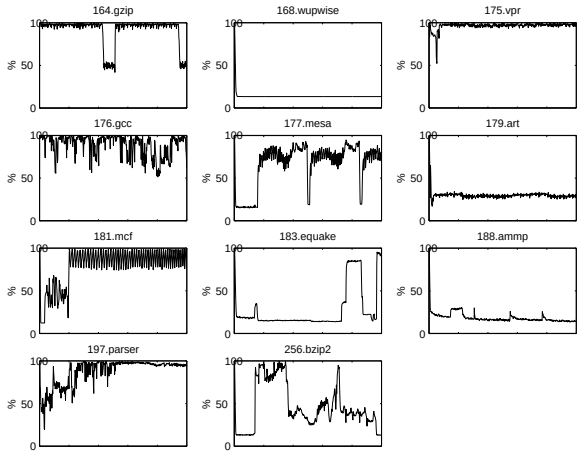

Figure 6: Different effective cache sizes during different phases of a 32KB data cache determined by MTR. The x-axis represents 0 to 1.5 billion cycles.

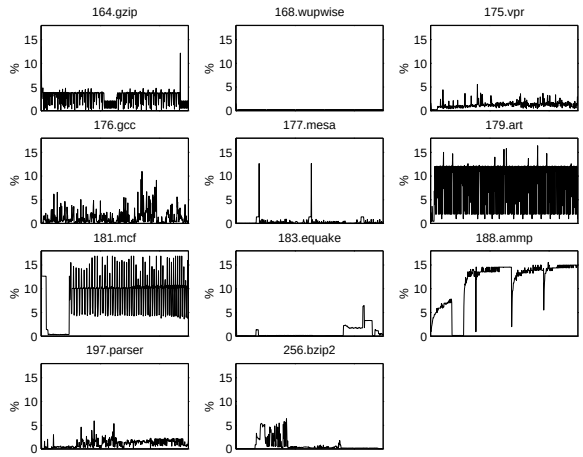

Figure 7: Cache miss rates during different phases of a 32KB data cache determined by MTR. The x-axis represents 0 to 1.5 billion cycles.

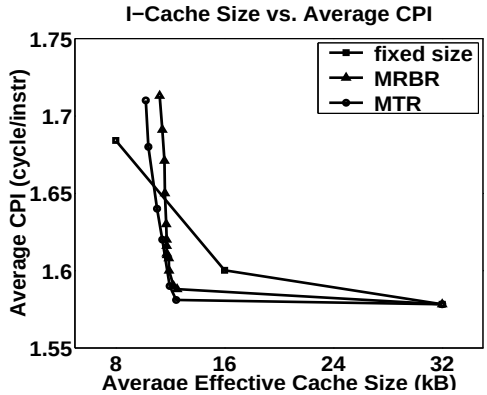

Figure 8: CPI versus effective cache size for L1 instruction cache. MTR and MRBR have similar performance.

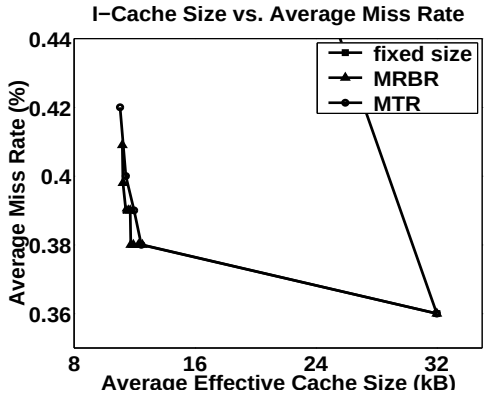

Figure 9: Miss rate versus effective cache size for L1 instruction cache. MTR and MRBR have similar performance.

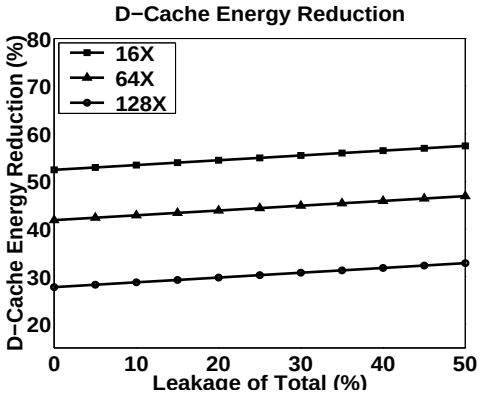

Figure 10: Data cache energy savings. X-axis represent the percentage of leakage energy of total energy. Y-axis represents savings. Each curve represents a different L2 access energy quantified as a factor of L1 write access energy.

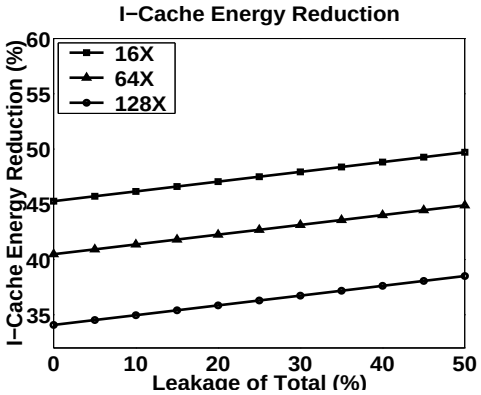

Figure 11: Instruction cache energy savings. X-axis represent the percentage of leakage energy of total energy. Y-axis represents savings. Each curve represents a different L2 access energy quantified as a factor of L1 write access energy.

Table 1: Energy components of CAM-tag cache in TSMC 0.25 μm technology.. A $\surd$ means the read or write access performs that operation, thus uses that energy component.

Operation	Energy (pJ)	Read	Write
CAM-Array Search	57.1	$\surd$	$\surd$
Data-Array Read	26.2	$\surd$	
Data-Array Write	53.5		$\surd$
Decoding & I/O	12.2	$\surd$	$\surd$
Total		95.5 pJ	122.8 pJ

while turning off the maximum number of partitions in the cache.

Each different curve represents the energy savings of a specific L2 access energy. We chose an range of L2 access energy, from 16× to 128× of the L1 write access energy. For data cache, MTR reduces energy by 28%, when there is no leakage energy and L2 penalty is 128× of L1 write access energy, to 56%, when 50% of the cache energy is leakage and L2 penalty is 16× of L1 access energy. Similarly, MTR reduction ranges from 34% to 49% for instruction cache depending on leakage percentage and L2 penalty.

7. CONCLUSION

In this paper, we presented MTR, a dynamic cache resizing technique for CAM-tag caches. The dynamic control mechanism of MTR uses a set of duplicate miss tags to keep track of the miss rate as if the entire cache was used. Resizing decisions are made according to the difference in the actual miss rate and the miss rate of the miss-tags. The control mechanism is only activated on misses, thereby saving energy and allowing the duplicate tags to be implemented in slower and denser logic using low leakage transistors. The cache partitioning mechanism saves both switching and leakage energy in unused partitions, and allows resizing at a single line granularity. The subbanks are resized independently in non-overlapping phases to avoid write back bursts. With around 10% area overhead, MTR reduces 28–56% of data cache energy and 34–49% of instruction cache energy, where the baseline caches were highly optimized for low-power but fixed-size operation.

8. ACKNOWLEDGMENTS

We would like to thank members of the MIT SCALE group for feedback and comments on earlier drafts of this paper. We also appreciate the comments from the anonymous reviewers. This work was partly funded by DARPA award F30602-00-2-0562, NSF CAREER award CCR-0093354, and a donation from Infineon Technologies.

9. REFERENCES

[1] D. Albonesi. Selective cache ways: On-demand cache resource allocation. In *MICRO-32*, November 1999.

[2] B. Amrutur and M. Horowitz. Techniques to reduce power in fast wide memories. In *ISLPED*, pages 92–93, October 1994.

[3] B. Amrutur and M. Horowitz. A replica technique for wordline and sense control in low-power SRAMs. *IEEE JSSC*, 33(8):1208–1219, August 1998.

[4] N. Bellas, I. Hajj, and C. Polychronopoulos. Using dynamic cache management techniques to reduce energy in a high-performance processor. In *ISLPED*, pages 64–69, August 1999.

[5] D. Burger and T. Austin. The SimpleScalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[6] Cadence Corporation. `http://www.cadence.com/`

[7] K. Ghose and M. B. Kamble. Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation. In *ISLPED*, pages 70–75, August 1999.

[8] H. Hanson *et. al.* Static energy reduction techniques for microprocessor caches. In *ICCD*, May 2001.

[9] H. Zhou *et. al.* Adaptive mode control: A static-power-efficient cache design. In *PACT*, September 2001.

[10] K. Inoue, T. Ishihara, and K. Murakami. Way-predicting set-associative cache for high performance and low energy consumption. In *ISLPED*, pages 273–275, August 1999.

[11] S. Kaxiras, Z. Hu, and M. Martonosi. Cache decay: Exploiting generational behavior to reduce cache leakage power. In *ISCA-28*, June 2001.

[12] R. Kessler. The Alpha 21264 microprocessor. *IEEE Micro*, 19(2):24–36, March/April 1999.

[13] J. Kin, M. Gupta, and W. Mangione-Smith. The Filter Cache: An energy efficient memory structure. In *Micro-30*, December 1997.

[14] L. Clark *et. al.* An embedded 32-b microprocessor core for low-power and high-performance applications. *JSSC*, 36(11):1599–1608, November 2001.

[15] M. Powell *et. al.* Gated-Vdd: a circuit technique to reduce leakage in cache memories. In *ISLPED*, July 2000.

[16] M. Powell *et. al.* Reducing leakage in a high-performance deep-submicron instruction cache. *TVSLI*, 9(1):77–89, February 2001.

[17] S. Heo *et. al.* Dynamic fine-grain leakage reduction using leakage-biased bitlines. In *ISCA-29*, Anchorage, Alaska, May 2002.

[18] S. Narendra *et. al.* Scaling of stack effect and its application for leakage reduction. In *ISLPED*, pages 195–200, 2001.

[19] S. Santhanam *et al.* A low-cost, 300-MHz, RISC CPU with attached media processor. *IEEE JSSC*, 33(11):1829–1838, November 1998.

[20] S. Yang *et. al.* Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *HPCA-8*, Feburary 2002.

[21] Taiwan Semiconductor Manufacturing Company. `http://www.tsmc.com/`

[22] L. Villa, M. Zhang, and K. Asanović. Dynamic zero compression for cache energy reduction. In *MICRO-33*, 2000.

[23] M. Zhang and K. Asanović. Highly-associative caches for low-power processors. In *Koolchips Workshop, MICRO-33*, December 2000.